CSDL-T-1276

CACHE ANALYSIS IN A MULTIPROCESS ENVIRONMENT USING EXECUTION DRIVEN SIMULATION

by

John Hamilton Fraser III

August 1996

Master of Science Thesis
Northeastern University

The Charles Stark Draper Laboratory, Inc.
555 Technology Square, Cambridge, Massachusetts 02139-3563

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	9 Jan 97	

4. TITLE AND SUBTITLE
Cache Analysis In A Multiprocess Environment Using Execution Driven Simulation

5. FUNDING NUMBERS

6. AUTHOR(S)
John Hamilton Fraser III

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Northeastern University

8. PERFORMING ORGANIZATION REPORT NUMBER
96-121

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
DEPARTMENT OF THE AIR FORCE
AFIT/CI
2950 P STREET
WPAFB OH 45433-7765

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION AVAILABILITY STATEMENT
Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

14. SUBJECT TERMS	15. NUMBER OF PAGES
	181
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT

Cache Analysis in a Multiprocess Environment Using Execution Driven Simulation

A Thesis Presented By

John Hamilton Fraser III

to

The Department of Electrical Engineering

in partial fulfillment of the requirements
for the degree of

Master of Science

in the field of

Electrical Engineering
(Computer Engineering Concentration)

Northeastern University
Boston, Massachusetts

August 30, 1996

Abstract

Cache memory is commonly used to bridge the gap between microprocessor and memory speeds. A wide variety of cache designs are possible, so some method is required to evaluate the benefits and costs of the various alternatives. Trace driven simulation is commonly used by the computer architecture community to analyze potential designs. Traces of benchmark execution are applied to a model of the design under study. Most of today's computer systems have been optimized based on results of these studies.

One important aspect that is frequently ignored in trace driven studies is the effect of the operating system and multiprogramming on cache performance; most traces consist only of a single program's execution. It has been acknowledged in the past that this overhead introduces interference which limits the benefits of new designs, but evaluations using multiprogrammed traces have been neglected due to the lack of readily available tools that can capture such traces.

In this research we describe a new tracing system that allows the capture of both operating system and multiprogrammed execution data. Cache performance is studied using multiprogrammed traces of the SPEC benchmarks. We study the effects of considering multiple tasks on the cache miss rate. The performance variation is primarily due to the presence of context switches. In an attempt to extend this work, we develop an analytical model that is used to synthetically incorporate context switches into a single process' trace.

We have found that the operating system introduces a small but persistent overhead to cache performance. Additional processes have an even greater impact, which increases as the level of multi-tasking increases. Spatial locality is not significantly affected by these conditions, but the temporal locality of a program is substantially reduced by the presence of context switches.

Contents

List of Figures

List of Tables

1 Introduction

The technological improvements in processor technology are far outstripping the advances made in memory circuit design. As processors execute faster and faster, the latency experienced when accessing memory becomes a major limitation. Faster memory is available, but at greater cost. An economical balance between performance and price is achieved through the use of memory caches. The main memory is implemented using less expensive but slow technologies such as SRAM, making a large memory feasible. A much smaller memory cache is constructed of faster (and more expensive) memory circuits, such as DRAM, to be used as a buffer between the main memory and the processor. Sections of the data stored in main memory are copied into the cache, allowing it to be accessed much more quickly. Which sections of memory are copied into the cache, and how the information is maintained, is a function of the design of the cache [22, 36, 52].

A cache is effective in reducing the average memory access time because of certain properties found in software. The collection of instruction and data addresses used by a program over some time interval is referred to as its working set [3] or footprint [56]. The working set may change as the program executes, but it generally exhibits two properties:

1. spatial locality, and

2. temporal locality.

Spatial locality refers to the property that addresses tend to cluster together in space. References may be sequential or in some other way structured, denoting a high degree of spatial locality. Similarly, temporal locality refers to the property that addresses tend to cluster together in time. Addresses in the working set may be used repeatedly during their lifetime, denoting a high degree of temporal locality.

These two properties allow caches to improve memory system performance. A memory reference which is not in the cache causes a cache *miss*. The data at the referenced location and some number of its adjoining locations is brought into the cache. Due to locality, it is likely that either the same location (temporal), or nearby locations (spatial), will be referenced in the near future. When these references occur, they are already present in the cache and a cache *hit* ensues. On a hit, the data can be very rapidly supplied to the processor, much faster than an access to the main memory. The improvement provided by a cache becomes a function of how often a hit occurs

and how fast the addressed data can be provided to the processor, balanced by the delay introduced when servicing a cache miss.

The critical nature of caches has led to extensive study of various designs, configurations, and enhancements, all oriented towards increasing cache performance. There are diverse methods available to assess the alternatives, ranging from prototyping to simulation. Regardless of the method, the accuracy of the evaluation is paramount. The criteria used to justify any evaluation must accurately reflect the environment to which the cache will be subjected, otherwise any conclusions are questionable.

One of the major shortcomings of the most common evaluation methods is that the effect of the operating system and multiple user processes being executed are neglected. The methods are simpler, but ignore a major aspect of the computer's architecture. Several past efforts have shown the related impact is significant enough to warrant inspection [1, 2, 8, 11, 12, 41], and is certainly a more realistic representation of the execution environment. The drawback is the difficulty of incorporating these considerations into the evaluation. There is generally some overhead required, in time and/or resources, to perform such complex tests.

The research described here focused on developing a tool to capture multiprocess state information and perform subsequent evaluations, exploring its capabilities with studies in both detailed cache simulations and testing an analytical model. This thesis is organized as follows. In section 2 cache performance and evaluation methods are reviewed. Section 3 describes the analysis tool ATOM, and how it can be used specifically on the operating system and in a multi-process environment. Section 4 discusses the methodology followed in this research and outlines the tests performed. Section 5 reviews the results of simulations performed in the multi-process environment. In section 6 an analytical model is presented that can be used to simplify simulations with minimal loss of accuracy, which is tested in section 7. Section 8 concludes the work, with a summary of its contributions in section 9. Last are section 10, the acknowledgments and section 11, the bibliography. Two appendices are attached, A, copies of the programs used in this research, and B, tables of all simulation results.

2 Background

2.1 Cache Performance

Cache performance encompasses a variety of issues. At the most basic level, the performance of a cache can be defined by its miss rate (or ratio), the percentage of references applied to the cache whose data was not already present in the cache. Alternatively the hit rate, which is the percentage already present, may be referred to. The two values represent equivalent information, since the miss rate equals one minus the hit rate and vice versa. Depending on the system and evaluation performed, however, this metric may be an oversimplification. The goal of the cache is to improve the average memory access time, which is a function of more than just the miss rate. It is entirely possible for a cache to have a low miss rate, but due to other consideration have a long access time thus limiting its usefulness. Hence many evaluations are based not on miss rates, but rather refer to the cache latency [7, 8, 41, 47]. The drawback is that to perform an evaluation of that magnitude is much more difficult and requires modeling a greater portion of the system under test, so focusing simply on miss rates is frequently used anyway.

Regardless of the standard used, the cache miss rate is important, as the average access time does depend on this value. To understand the significance of the miss rate, it is important to understand the various sources of misses. A program generates a stream of memory references as it executes, which are applied to the cache. Cache misses are caused when an address in the reference stream is not present in the cache. This can occur for basically three reasons [3, 55]:

Start Up The first form of miss is caused the first time that a particular address is referenced in the stream. Since it has not been referenced before, there is no expectation that that memory location would have been copied into the cache. Such misses are encountered primarily when a program begins executing and all references are new, also called the warm up phase of the cache. The size of the cache and the program both contribute to the length of this phase. As the working set changes, additional start up misses are encountered as new locations are referenced.

Though a certain address may not have been previously referenced, it is still possible that its data is already in the cache. When data is copied from memory to the cache, it is moved in quantities called blocks. A block is usually larger than a single memory access, so a single miss fetches more data than is required for a single access. If a location is referenced that resides in

a block already fetched, it will hit, even though that particular address may be new. This is only effective for memory references that are primarily sequential, such as instruction fetches, in which case a large block size is beneficial. Footprints with less locality, such as data loads and stores, can actually have the reverse effect as large blocks bring in excess data which is never used.

Another technique to prevent start up misses is the use of prefetching [14, 15, 52]. This is essentially an attempt to predict what locations will be referenced in the near future, and fetch them into the cache before they are requested. The method of prediction can be hardware or software based, and must be accurate for prefetching to be effective. If data is falsely predicted and fetched into the cache, it may overwrite "live" data (live meaning that it is still part of the current working set), causing cache pollution. Additional enhancements such as a pre fetch buffer filter or victim cache can be used to limit this impact [22]. Using prefetching can improve miss rates, however it also increases the traffic between the cache and memory. An accurate evaluation cannot consider only miss rates with this technique, otherwise its drawbacks will be obscured.

Capacity The second form of miss is due to the finite cache size. A large program cannot possibly fit its entire working set into a small cache. As various parts of the working set are used, they will overwrite other live data. The obvious solution is to use a larger cache, but at additional expense. Another potential solution is to analyze the locations used in the working set. The references may cluster around certain blocks while others are unused. Changing the mapping of addresses to cache lines (or indices) may allow the references to be better distributed across all cache lines [7]. This technique is also an effective counter for the next type of miss, which together with capacity misses are sometimes referred to as *intrinsic interference.*

Conflict The third form of miss is due to conflict between two references. If two addresses in the working set map to the same cache line, each time they are referenced a cache miss may result (depending on the actual pattern of references). Again, altering the mapping algorithm may reduce the amount of conflict in a given reference stream by spreading out clumps. Another option is to use an associative cache [22, 52]. In this form of cache, each cache line (sometimes called set) can maintain multiple blocks, so multiple locations can map to the same line without conflict. The number of blocks held in each line is referred to as the set size or associativity

of that cache, and can vary from 1 to the maximum possible given the available chip area. This type of cache can be pictured as a two dimensional array of blocks, with the vertical dimension the number of lines and the horizontal the associativity. The bounding cases are a direct mapped cache with an associativity of one, and a fully associative cache with only one line. The drawback is that for a finite cache area, increasing the associativity decreases the number of cache lines, so each line in the cache has more locations mapped to it and a corresponding heavier load. Also, associative caches are frequently slower, which should be a factor in comprehensive evaluations.

These three categories comprise the basic types of misses found in a process' reference stream. They must be considered in even a minimal performance measurement, although there are other cache components that may improve memory system performance without affecting the miss rate.

Other cache enhancements which do not directly affect miss rates are usually related to access times. Techniques such as using a Translation Lookaside Buffer (TLB) [49] can perform cache lookups and virtual address conversions in parallel. Other methods include using hierarchies of caches, such as a small direct mapped cache on chip and a second level larger cache, possibly associative, off chip. Using combinations of caches can potentially improve the performance more than a single highly complex cache [52]. In some instances an entire cache is not added, but various buffers or filters are accommodated, such as the prefetch buffer or victim cache [7].

The cache performance will depend on many characteristics of the cache. Some of the most basic are its size and structure, and the method it uses to resolve both hits and misses for each reference type (instruction fetch, data read, and data write). Performance enhancing mechanisms may also be included, each addressing various deficiencies. Studies have shown that multiple mechanisms in concert are generally the most effective [47]. The wide variety of cache designs makes the ability to evaluate various options paramount, and there are concerns that have yet to be addressed which further complicate analysis.

So far in this discussion, caches have been considered in an idealized environment. Modern computers do not simply execute a single program continuously until its completion. The operating system generates its own references as system calls are requested. The operating system also generates references for processes such as interrupt services and other management tasks, which are performed periodically. Even more complex is a multiprocess environment, with multiple programs or threads being executed. In a multitasking system there are several processes or tasks all vying

for system resources, one of which is memory. In a uniprocessor system, control is accomplished by time sharing. The various tasks are executed for finite intervals and then execution is switched to another process — called a context switch. As each task is scheduled and executed, it generates its own reference stream with unique characteristics. The individual streams are interleaved by the context switches to yield an aggregate reference stream which impinges on the cache [19, 31, 56].

This introduces a new mechanism causing a fourth and final type of miss, *transient* cache misses. When a process is swapped out during a context switch, the process or processes that execute until the original process is returned will overwrite its cache data. This data may still have been live, so the overwrites may cause additional cache misses once the original process is restored. This is referred to as *extrinsic interference* [2], as opposed to the intrinsic interference discussed above, and can be thought of as a reload period after each context switch as evicted data is returned to the cache [56]. The impact of extrinsic interference will magnify with increased multiprogramming as the duration of each swap is extended, although this can be partially negated by stabilizing the time quantum that each process executes.

Some designs call for the cache to be totally flushed (invalidated) at each context switch automatically. This might be appropriate for a control mechanism such as the cache type structure used to implement a TLB, but in an instruction or data cache it is quite likely that some of the live data from a process would still be resident when that process returns to execution. By maintaining the cache data for as long as possible, the extrinsic interference is kept to a minimum; although this does require additional overhead to monitor the owner of each line of cache data, and complicates analysis [22].

Other architecture issues can further complicate performance consideration. A multiprocessor system is similar to what has already been discussed, but more complicated. Not only are multiple reference streams being generated, they are generated simultaneously and possibly applied to multiple caches. Each processor may maintain its own memory structure or they may share a common structure. This raises the issue of cache coherency, or the property that data stored in memory is properly maintained in each location it is represented. If multiple processes share memory but have their own caches, care must be taken to monitor when data is in multiple caches (shared) so that if the data is modified, it is modified in all caches. Various policies can be used when data is stored to the cache, such as write through, meaning data is written to memory as soon as it is written to cache, or write back, meaning the data is not written to memory until it

is evicted from the cache. Each has various advantages and disadvantages, and in turn affects the policy used to maintain coherence [15, 29]. There are a variety of other technical issues as well, such as communication and synchronization, making this a very complex design. Even more radical departures from the traditional von Neumann architecture, to a dataflow architecture for example, cause even greater difficulties in defining evaluation criteria [30].

2.2 Cache Analysis

2.2.1 Methods

There are a variety of methods available to evaluate cache performance. General reviews are presented in [1, 11, 13, 60]. The techniques can be broken down into various categories:

Analytical Models The most abstract form of analysis is based on a theoretical prediction derived from the test system's characteristics and assumptions of how it is loaded. Developing a model of the system under test requires certain assumptions which may oversimplify aspects of cache design, neglect relevant characteristics of the input, or may not be sufficiently verified to warrant their use. The accuracy of the evaluation is limited by the accuracy of the theoretical model, and unfortunately, the more accurate and comprehensive the model, the more difficult it is to solve [3]. Some models are based on abstract parameters with little relation to the actual system [31], and others may require considerable test program characterization; to the point that other methods would be equally suitable [56]. The most successful models tend to focus on very limited aspects of memory system performance to reduce their scope [28, 55].

Hardware Evaluation The antithesis of theoretical analysis is hardware evaluation. In this method, the test system is implemented and inserted into some platform. Its performance can then be monitored directly as the platform is operated. The actual analysis is quite quick, as the processing is conducted at the same speed as the platform, however the test system must be constructed, which may be a slow and expensive process. The other disadvantage is that to test a variety of alternative designs, each alternative must be constructed. This limits the flexibility and can be even more costly. Rapid prototyping can make this method more attractive, and some examples have been found in [11, 24]. Using techniques of hardware emulation can also be more efficient, although they are slower [40].

Trace Based Simulation By far the most common form of analysis is trace driven simulation. A trace of program references is generated and applied to a model of the system being tested. The model is simulated in software, and can be as complex as accuracy dictates. A software model is very flexible, but simulations are slower to compute. Also, the traces must somehow be stored, which requires a great deal of memory, although they can be reused. The trace can be as complex as desired, and there are a variety of methods that can be used to generate it:

Synthetic Generation Workloads can be created for system test through the use of synthetic generators. No programs need be executed, reference streams are simply generated randomly. Some control is provided through defining random variables and their distributions, establishing the desired characteristics of the workload. Since it is artificially generated, however, its accuracy is highly suspect. Various examples of this technique can be found in [35, 46, 57, 58].

System Emulation Another alternative which does not require program execution uses system emulation. A test program is required, but it is fed into an instruction set simulator which generates reference stream data. This pseudo execution of programs is very slow, though, and is rarely used [60].

Hardware Capture The last two methods monitor the execution of a test program on some platform, capturing the reference stream as the program executes. In hardware capture, the platform is modified so that as it executes the test code, the references generated are collected and stored. It is easy to capture a wide variety of references in the trace working at this level, but this technique suffers from the disadvantage of requiring unique hardware and/or costly modification. The two most common forms of hardware capture have been accomplished by modifying the microcode of the CPU [1, 2], or by using test probes inserted into the system to electrically read the system status [11, 60]. The first can only be used with certain architectures, however, and the latter is limited by the external visibility of data (for instance, an on chip cache could not be monitored). Once each reference is captured, there are a variety of ways to record it, such as storing it in a buffer and occasionally writing the buffer to a file. The method must be able to record data as fast as the system generates it, which may be a significant limitation. Despite the disadvantages, this method is frequently used in certain situations where other methods may not be feasible, such as very complex architectures [5, 59].

Software Capture The most common form of trace generation is by software capture. Instead of modifying the testbed, the software can be altered so that information about the program's execution is recorded. Again, the trace is generally stored in a buffer until it can be written out to a file, although there are alternatives. Software capture is more flexible than hardware based methods, as the information that is collected can be easily updated as evaluation needs change, but capturing all aspects of the reference stream (such as the operating system) can be difficult. Capture can be based on snooping programs [50], interrupt generation [32], or by explicitly modifying the test code. This modification can occur during compilation [7, 8, 25, 43, 45] or can be applied to an existing executable [11, 12, 13, 54].

Extensions There are also various extensions that can be used with the above techniques to improve their efficiency. For instance, one major drawback of trace based simulation is the storage space required for the traces. To compensate, it is possible to have the analysis program executing concurrently with the trace generation, so that no long term storage is required; one example is [8]. This does preclude reuse, however. Other techniques include sampling traces to reduce their length, although this may affect their accuracy depending on what assumptions are made in the sampling process [1, 2, 6, 33, 61]. It is also possible to simply compress the trace file, but this is only a short term solution. Other extensions include using various processing algorithms such as stack based processing to simplify simulation [48, 64], or reducing processing time with parallel computation [42, 43, 63]. Analytical models can be used in conjunction with program traces to simplify simulation and provide evaluation over a variety of system characteristics with a single execution [3].

2.2.2 Issues

The evaluation method used must accurately reflect the type of workload that would be present in a real system. This is particularly a concern when analytical models are used, as programs may not be executed at all, so a statistical approach is common [57, 58]. For hardware measurement and trace based simulation, this problem is addressed by selecting appropriate programs to be executed in the evaluation. Specific programs known as benchmarks are used as accepted standards for testing [34, 45, 49]. There are differences in workloads depending on the type of programs being considered, whether they are technical or commercial applications [37], so generally multiple test

programs are used to ensure the evaluation is comprehensive. The better test programs will have a large and complex footprint to exercise the cache fully, although this can make standardization more difficult and analysis slower.

Once a workload is identified, how it is represented and used in the analysis can vary. If a program is executed or traced, there are a variety of concerns that must be addressed for the evaluation to have much confidence [1, 11, 13, 60]:

Reference Scope The simplest forms of references to monitor are from a single process [7, 25, 45, 61, 62], but though they are easy to capture they are also not particularly a realistic reflection of cache loading. Even in this basic form, care must be taken to ensure that shared libraries and other common structures are captured. A more realistic reference stream includes additional processes, and if possible, the operating system. Hardware evaluation of a cache and hardware based trace capture for simulation do allow capture of all references, but as mentioned before they have other drawbacks. It may be difficult to identify the source of particular references, too, making analysis more difficult. Through the use of comprehensive software capture mechanisms, it is possible to capture traces with multiple processes [8, 41]. In its most complex form, this mechanism can also be used to capture traces that include the operating system [1, 2], however a thorough understanding of the test system is necessary for proper implementation. Such references are more difficult to capture, and present a new problem in processing. The multiprocess environment is non-deterministic, the reference stream can vary even for execution of the same test programs as scheduling and interrupts change the execution pattern. For a truly accurate comparison, all tests must be performed from a single stored trace, or they must all be performed concurrently from the stream as it is generated and processed [8].

Reference Length Another accuracy problem with reference streams are their length. As caches increase in size, more references are required to fully exercise them. A large cache can contain a large footprint, so a long program is needed to generate such a footprint. This is particularly relevant for RISC machines, which will have significantly longer traces for a given program because of the increased number of instructions. Current practices call for on the order of 100 million to 10 billion references to be an adequate [8]. Hardware evaluation places no constraint on program execution, but traced based methods may be limited. Early tracing mechanisms

could not generate long enough traces, so shorter traces were stitched together [1, 2]. In other cases, single process traces were interleaved to approximate a multiprocess environment [56]. Recently, more robust methods have become available so that such artificial measures are not required [13, 20]. Long traces are difficult to manage because of the storage space they require. Analysis can be conducted on the fly so the traces are used as they are generated [8], or the traces can be sampled to reduce their length [3].

Platform Impact The operating system and compiler used affect cache performance. The relative location of a program's instructions and data will affect the amount of conflict since those locations determine which cache line each will be mapped to. Other considerations such as data alignment, prefetch/flush commands, and program scheduling will also affect the reference stream. The compiler generates code optimized for a certain physical memory system, so may not be ideal for the test memory systems being considered. For the purposes of most evaluations, this effect is considered to be equivalent across all designs, and can be ignored, particularly by using the least optimized code possible [69].

The memory system used on the platform will also affect the evaluations performed with it. The size of the memory can produce page faults and other activities, which in turn generates additional overhead references that would not have occurred in the modeled system. Other systems may dynamically schedule activities based on the system state, which may include memory system performance, so ordering of events may be subtly altered.

In certain architectures, the scheduling of references is linked directly to the memory system performance. For instance, one possible method to hide the cache latency is to generate a context switch on any cache miss. For this to be viable, the overhead of performing a context switch must be less than the latency to service a cache miss. If this is the case, the cache performance then plays a major role in defining the reference stream. One solution used in [38] is to not only simulate the cache, but the pipeline and instruction set as well. The test program executable file is fed into the simulation which executes it "virtually". Such a simulation is very comprehensive but also quite complex. Parallel systems present a similar problem. References may be generated for one system and a variety of memory configurations can be tested, but any changes to the architecture of the underlying system may totally invalidate the accuracy of the reference stream. Also, multiple reference streams are being

generated simultaneously, either being applied to the same cache or multiple caches that must remain consistent. Generally, such complex architectures dictate certain types of evaluation methods, using either synthetic [46] or hardware monitored traces [59] for analysis. Another option is to capture robust traces with more information than just simple addresses so that the execution stream can be re-created for a variety of systems [26, 32].

Reference Mapping When a reference is applied to the cache, it is mapped onto a cache line. A simple hashing of the address bits may be used, or a more complex algorithm, possibly including other information such as the process identifier [52]. The algorithm can vary with the system and depending on how addresses are collected it may be relevant. Depending on the capture method, the addresses generated may also be virtual or physical. Virtual addresses may be used to model caches, however this is a simplification. The actual memory system must at some point convert all addresses to physical form. This conversion affects how lines are mapped from memory to the cache, so it is relevant to cache performance. Unfortunately, converting to physical addresses is a very complex task that requires considerably more system state information than is provided by a basic reference trace. Since the placement of programs in memory affects their mapping into the cache, the loading of programs into memory is also relevant, although this is usually controlled by the operating system.

There are additional concerns relevant to particular methods. If traces are captured, care must be taken so that the act of tracing does not affect the trace generated. Hardware capture methods tend to be non-intrusive, but have other drawbacks. Software based methods in particular are very intrusive since they modify the test programs, and certain measures must be taken to compensate [1, 11, 13, 60]:

Address Skewing The code added to a test program will change the various address used for both instruction fetches and data accesses. If the addresses during execution are used directly for the analysis, the results will be skewed. Instead, the addresses must be calculated based on what the reference position would have been without tracing. This is normally handled by the trace generation software, and can be transparent to the simulation model.

Processing Skewing The additional code inserted into a program can also cause the processing characteristics of the test program to be skewed. The added code may make additional calls to system resources or generate additional interrupts. The capture mechanism should ideally

identify the source of references so they can be discarded if not generated by the original test program, although this is difficult when the operating system is considered.

Program Size Since program size is increased, certain aspects of execution will be changed such as paging. The larger programs will occupy more memory and hence require greater system overhead to manage.

Program Speed The program speed is related to the program's size. The additional code introduced into programs can easily slow down their execution by an order of magnitude [8]. The more processing introduced by tracing, the greater the slow down will be. This affects the accuracy of traces in two ways. Longer programs will have a disproportionate number of real-time interrupts during their execution. Some form of scaling must be used so the frequency of this type of interrupt is reduced within the trace. Neglecting to perform the service routine is possible, however may affect system performance. The longer programs will also have a disproportionate number of context switches as the additional code can both cause switches as well as slow down the original program so that less is accomplished during the maximum execution interval allowed by the scheduler.

Once such concerns are addressed for a given evaluation methodology, an analysis can be performed with a great deal of confidence in its results.

2.3 Current Work

As early as the late 1980's, the impact of the operating system and additional processes was recognized as a concern in memory system performance [1, 2, 3]. More recent work has consistently validated the supposition that this impact was significant enough to warrant further study, and should be included in any comprehensive memory system evaluation [5, 11, 12, 13, 41, 59]. More importantly, as computing capability increased, it has become possible to capture longer and more complete traces directly, without using such patch work measures as described before.

Much of the recent work has revolved around trace driven simulation with software capture methods. Many studies still consider cache performance, although others are becoming more focused, looking at specific areas such as the effect different operating system structures can have on memory system performance [11, 12]. Some of the methods used are either proprietary [37], or especially designed for a certain application [62]. Some generic tools have been generated, such as Epoxie,

which rewrites assembly code to generate address traces [11, 12, 13].

Another such tool is ATOM, very similar to those found in [11, 12, 13, 37]. Developed by DEC's Western Research Laboratory, ATOM is a general purpose program analysis tool that can be customized to perform a wide variety of different evaluations. Until recently, ATOM focused on only the single process environment, but in its latest versions, it now has the capability to capture traces that include the operating system as well as multiple user programs. This research has revolved around refining this capability and demonstrating its applicability to cache analysis.

3 ATOM Overview

3.1 General Use

ATOM (Analysis Tools with OM) [51] is not a specific application; rather it is a toolset that can be used to produce custom analysis tools. It provides the framework to generate program traces during execution and pass the trace data to analysis routines through a procedure call interface. The analysis or simulation program is actually incorporated into the test program, so as the test program is executed, so is the tool. This procedure is commonly referred to as *execution driven simulation*, effectively combining the act of tracing and analysis. Tracing of this type alleviates the need for trace storage, as well as the difficulties of synchronizing a separate analysis program with the test programs.

The analysis performed can vary a great deal due to the flexibility provided by ATOM. Tracing is performed on selected events such as program start/stop, basic block boundaries, memory reads and writes, instructions, or procedures. Certain types of a given event can be selected (i.e., a certain procedure call), or all instances of an event (i.e., every instruction). The trace capture is inserted as a function call to an analysis routine, so that when a particular event occurs during execution, information about that event is passed to the analysis routine where the event data is recorded, processed, or in some other way used to perform the desired evaluation.

Given this type of framework, tools are quite easy to generate. For a simple cache simulator with a single process, the test program is instrumented at every instruction fetch and at every data load or store. The memory location referenced by each instruction is passed to the analysis routines corresponding to that reference type. Within the analysis routine, the cache simulation is performed, so that when the test program concludes, the simulation is completed.

The specific form of analysis to be "instrumented" into the test program is incorporated at link time by ATOM using two files:

1. the *instrumentation file*, which instructs ATOM which events to trace on and what event information to pass to the analysis routines, and
2. the *analysis file*, which defines the various analysis routines and any other subsidiary functions required.

It is a very simple process to use. The test program is compiled, and then used as input to

the ATOM program with the following example command line:

```
%atom program.rr inst.c anal.c -o program.trace
```

The program is then executed and the desired analysis specified by `inst.c` and `anal.c` is performed. This is a very simple example. There are various control flags that ATOM accepts, these are described in both the on-line documentation and the program manuals.

For simplicity it is also possible to define tools for ATOM. A tool description file is created which specifies which instrumentation and analysis files to use, as well as the various flags to pass to ATOM. The programs are instrumented with a tool by using the command line:

```
%atom program.rr -tool eval -o program.trace
```

In addition to simplifying the command line, defining a custom tool also allows additional control flags to be used. The basic ATOM command line does not accept loader flags, for example, so the flags necessary to include shared libraries such as `math.h` (`-lm`) cannot be used. This would normally prevent analysis routines from accessing such basic functions, which is obviously an inconvenience. By defining a tool, it is also possible to define additional flags and at which stage of instrumentation they should be used - allowing the use of shared libraries and other linker/loader flags.

With the flexibility provided, ATOM is a versatile tool, but accuracy is still a potential problem. Another strong point for ATOM is its robustness. In the cache example above, one major concern is the fact that by adding additional code to the program, the reference stream becomes skewed by the additional instructions. This is automatically compensated for by ATOM during instrumentation, so that the addresses passed to the analysis routines are those of the memory references without tracing.

Another area ATOM excels in is its care with shared libraries. Many simulations totally neglect shared libraries, which may be a significant portion of the code depending on the application. Programs can be compiled with the non_shared option, or ATOM can instrument the shared libraries as well. To be even more exact, an instrumented and non-instrumented copy of the shared library routines are produced. This way if the instrumented program calls a shared library, the instrumented version of the library is used. If the analysis routine calls the same library function, the non-instrumented version is used so that the analysis is not corrupted.

Until recently, ATOM was not capable of tracing the operating system, and was not partic-

ularly suitable for tracing multiple test programs. The latest version of ATOM, however, does allow instrumentation of the operating system. The initial tests of this facility were performed by Eustace and Chen in [20], but some aspects were not particularly well addressed. The primary focus of this research has been to further test and build on their work [24].

3.2 Operating System Implementation

With the latest version of ATOM, it is now possible to instrument and study the operating system, specifically the OSF kernel. It is treated much as any program would be, albeit a very large and complex one. Because of the unique nature of the operating system, there are certain measures which must be taken that are not required for a normal program. Part of the mechanism used to study the kernel is also used to capture traces with multiple user processes as well.

3.2.1 Set Up

To use ATOM with the operating system, some modifications are usually required to the test platform. More memory may be needed to execute the larger programs, 128MB is recommended by DEC. The larger programs will also require more swap space (256MB recommended), a larger user file space, and an expanded root partition (up to 60MB depending on the application). ATOM version 2.20 or later must be installed, with the WRL enhancement kit. Both are available from DEC via anonymous FTP.

Changes are necessary to allow the kernel to be instrumented. The `makefile`, normally in the `/usr/sys` directory, must be modified and the kernel remade. The two modifications required are:

1. The `LDFLAG` line must have the `-ncr` flag removed. This flag removes the compact relocation records, and is not compatible with ATOM.

2. The `ALPHA_TEXTBASE` must be increased to account for the larger kernel size. This value represents the amount of space in memory allocated for the kernel text, usually set at `h230000`. Instrumentation increases the size of the kernel so this value must be increased accordingly. The required increase will vary, so occasionally the kernel must be generated twice. First a rough estimate of the necessary increase is used to make a kernel which is instrumented. The `nm -B` command can then be used to calculate the actual value needed. If it is too small, the

kernel will crash, and if it is too large, memory may be wasted. For the work performed here, a value of h2C00000 was used.

Once the makefile has been altered, a new kernel is created by the sequence of commands:

```
#make clean
#make depend
#make
```

These commands must be executed as root; using the sudo utility is not possible as the kernel will not be made correctly. During testing it was useful to have multiple kernels available with different ALPHA_TEXTBASE values as needs changed. If multiple kernels are made, it is necessary to rename the existing kernels before a new one is created as all existing files of the form `vmunix*.*` are erased during the make process. The new kernels are then instrumentable as any other program.

3.2.2 Programming

The act of instrumentation inserts function calls into the test program. These functions are executed as each event is reached during program execution, performing the desired analysis. For a cache simulator, those events are instruction fetches, data reads, and data writes. At each memory reference, the address referenced is passed to the analysis function for processing in the cache model. Additional functions are used at program start and end to initialize the simulation parameters and report the simulations results. The various functions and the instrumentation are defined in the two ATOM files mentioned previously for both the kernel and test programs.

To incorporate the operating system into the analysis, it is necessary for the operating system and test program to share data. The cache state must be accessible to both programs, as well as other counters and synchronization flags. This sharing can be accomplished via the `/dev/kmem` or `/dev/mmap` utilities. The shared data is local to the kernel. When the test program begins, either of the utilities is used to map the shared data into the test program's address space, where it can be accessed via a pointer. Now the two processes have a common data structure that is the core of the simulation. To use these utilities, there are two requirements. First, the test programs must be run as root to access the `/dev/` files. Second, two copies of the kernel must be created. One is the executable which is actually loaded, the other is a debug version which contains the symbol table information necessary to perform the mapping. The debug version stays in the same directory as

the test programs.

The ability to share data is the also key to capturing traces from multiple processes. As described above, data is captured from two processes, the kernel and the user program. As will be seen, the same technique can be used to increase the number of processes being captured. The example above uses shared cache state data, but any set of data may be shared to provide the desired capture information.

The instrumentation and analysis files are not substantially different for the kernel and user programs. For the kernel, a test must be used to ensure that certain procedures are not instrumented (see below). For the test program, the shared data must be mapped at program start and the data recorded at program end. Otherwise, the analysis functions may be more or less the same. For the cache simulator, a process identification value is passed with the address so that the sending process is recognizable.

Figure 1 shows logically how the original code and analysis routines work together to perform the desired analysis, in this case the cache simulator.

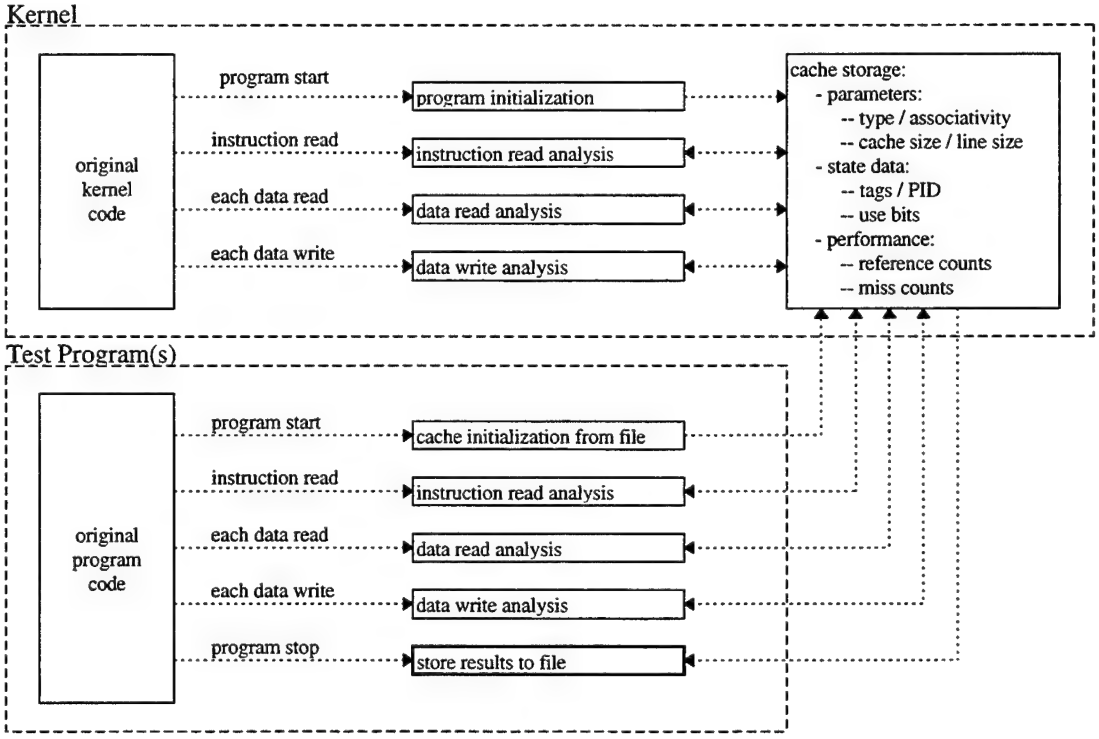

Figure 1: Program Block Diagram

3.2.3 Execution

Once the required files are written, the implementation is not substantially different from that of any other test program. The two instrumented versions of the kernel are produced with two slightly different command lines. For the executable:

```
%atom vmunix kern.inst.c kern.anal.c -Xkernel -Xgprog -o vmunix.trace
```

and for the debug version:

```
%atom vmunix kern.inst.c kern.anal.c -Xkernel -g -o vmunix.debug
```

The various test programs are also instrumented as described above. The executable version of the kernel is moved to root, and the system is restarted with the `#shutdown -h now` command. Using `boot -fl i`, the system is restarted and the instrumented kernel is specified and loaded. The testbed is frequently shutdown, so it was helpful to have a dedicated system for this research so that other work was not interrupted. Once the kernel is running at the desired execution level, the test programs are then executed normally, performing the analysis. It is recommended that a batch file be used to run test programs to simplify testing.

3.3 Problem Areas

3.3.1 ATOM Limitations

Certain characteristics of ATOM define limitations on the instrumentation which can be used within the Unix kernel.

- Since it is the operating system, tracing cannot be based on the program end event.
- Certain kernel procedures cannot be instrumented. These are the `locore`, `lockprim`, and `spl` libraries, which account for only 132 out of 10,678 kernel procedures so the error induced should be negligible.
- Floating point numbers cannot be used within the kernel.
- The ATOM model used when simulating dynamic memory allocation is not accurate within the kernel, so analysis of this aspect of program execution is suspect.
- No system call interfaces can be used within the kernel.

Most of these limitations are not particularly significant, although the last is inconvenient. Without system calls, file IO is not possible, which precludes using a file to set evaluation parameters. This makes it very difficult to dynamically define analysis parameters, so in many cases the programs and operating system must be re-instrumented for each desired evaluation (i.e. a separate run for each cache configuration). Many other shared library routines, such as mathematical functions, are also unavailable. As future versions of ATOM are released, hopefully some of these shortcomings will be addressed.

3.3.2 Kernel Limitations

Working with the kernel also entails certain problems, especially for a programmer unfamiliar with the operating system environment. The kernel is difficult to manipulate, requiring special access privileges. The critical nature of the program requires careful handling, although based on previous work, instrumentation errors will not damage the system — a kernel improperly instrumented will usually not even boot. The primary difficulty of working with an operating system is the difficulty in debugging. Most debugging tools cannot be used to debug a kernel, and many of the error messages generated are cryptic. Initial testing of instrumentation code should be done on generic user programs, and only when working on that level should it be attempted on the kernel. This provides better checking, and a much faster debug and test cycle. Working with the kernel is a slow process. Making a new kernel takes up to 8 minutes, and each instrumentation can take as much, if not more, time. Even assuming a new kernel is not required, to test a kernel usually takes about 20–30 minutes (as compared to the almost instantaneous results from a simple user program). Even with debugging on a user program, many problems will only appear in the kernel, so in general, development is very slow. Some of this may have been due to system limitations, but only a minor improvement should be expected with better resources.

There were three obscure errors found regularly during kernel testing:

1. `KSP INVAL`
2. `bootstrap address collision: image loading aborted`
3. `trap: invalid memory access from kernel mode`

The first error can occur when the kernel is loaded or during execution. This is roughly equivalent to a segmentation violation which is normally caused by a misuse of pointers. This error may

also be caused by running out of memory, if there is not enough stack or heap for the kernel to execute. The second message always appears during kernel loading. This is caused by an incorrect ALPHA_TEXTBASE assigned in the makefile. The nm -B command should be used to determine the correct value and the kernel remade. The final error always occurs during test program execution. This was an intermittent error and the cause was never found, even after conferring with DEC. The error always occurred in the kernel's thread_preempt routine which suggests it is related to interrupts and/or context switching. The error was linked to the size of the test programs being executed. A single large program could cause the error (such as Xlisp), or combinations of smaller programs (such as Alvinn with any other program, or Compress, GCC, and Espresso all together). Since it occurred with only one test program running, it cannot be caused by having two or more test programs sharing the kernel's data structure. The memory of the testbed was increased from 64 to 160MB with no effect. The hardclock scaling (see below) was reduced to its minimum value of 50% with no effect. To isolate the problem it will be necessary to complete an examination of the kernel which is beyond the scope of this work. The most likely cause is the threaded execution of the kernel and the lack of firm control within the analysis routines; although it is possible that the hardclock scaling is the culprit.

3.3.3 Program Size

One common problem with any software-based tracing method is the increase in program size. Since the program is instrumented with not only tracing information, but also analysis functions, this is a greater concern when ATOM is used. The normal OSF kernel is about 8–9MB. If the same kernel is instrumented with a function call at every instruction, and an additional call at every data read or write, the kernel will grow to 92.7MB and require an ALPHA_TEXTBASE of about h5A00000. A kernel this size could not even be loaded on the test machine. By instrumenting groups of instructions (and still each data reference), the kernel is only about 46MB with an ALPHA_TEXTBASE of h2C00000, which is executable. Instrumenting just instruction or data accesses will reduce the size by about half. It is important to note that the size of the instrumented kernel is primarily a function of the degree of instrumentation, not analysis. Changing the amount of analysis processing only varied the size of the kernel by about 4MB.

Besides the strain on the system from working with such a large kernel, it also raises an accuracy issue. The kernel used in our tests left only 15MB of memory available for test programs,

yet this is supposed to be simulating a system with about 50MB of free memory. The situation is even worse when the fact that each test program is also instrumented and significantly larger than normal is considered. Such large programs require more paging, which in turn skews the amount of overhead each program requires. For more accurate results, the amount of memory should be increased proportionately.

3.3.4 Execution Speed

Execution speed becomes critical when considering the instrumented kernel. The inclusion of tracing can reduce the execution speed of a program by an order of magnitude [8], more so with the additional processing. A slowdown of this magnitude may not be tolerated by the operating system. At some point, the kernel becomes so slow that it cannot function correctly. Interrupts and service requests may be generated faster than they can be serviced, effectively hanging the system during boot up. This can also be seen during test program execution if too many processes are executed — the kernel simply thrashes and the system stalls. Even assuming the operating system does work, basic tasks can take an inordinate amount of time. Booting a kernel with a basic cache simulator in multi-user mode and logging on took over an hour in one test. Several methods have been explored to accelerate the kernel and counter this problem.

The first is to use a different programming style for the kernel analysis routines. Only the bare minimum code necessary to perform the desired task is used. No additional function calls are made beyond the initial call to the analysis routine, eliminating extra switching. Any additional computation is incorporated into the primary function, even if this requires duplicating code. Loops should be used sparingly and the iterations minimized, and any other time consuming operations should be optimized. Minimizing data storage may help, but is not a primary factor. These techniques will definitely speed execution, particularly eliminating function calls, so even though some of these changes introduce poor programming practice from a software engineering standpoint, they need to be used.

If the kernel boots, but is too slow to execute the test programs in a multi-user environment, the first solution is to reduce the number of additional processes the kernel may be executing. Programs being run by other users or not part of the test should be eliminated. Other background processes associated with the operating system can also be killed. In multi-user mode, there are additional background processes executing, such as LAT, cron, network software, and printer daemons.

Many of these are not necessary for the tests and can be removed — the fewer processes running the faster the kernel will be.

If the kernel is still to slow, or will not boot in multi-user mode, it is possible to run the programs in single user mode. This effectively eliminates all extraneous processes and dedicates the system to the instrumented test programs. When the system boots to the first # prompt, do not start the higher execution level (the command is `^D`). The local disks can be mounted using `#mount -at ufs` so that the test programs can be accessed (assuming they are on a local disk). The simulations can then be executed normally. If multiple test programs are desired, they can be run concurrently by using background mode (`&`) for each. Using single user mode is significantly faster, and can be considered an advantage or disadvantage. It is true that most of the processes that would be executing in a "real" environment are absent, lessening the accuracy, however it also lets the analysis focus on the operating system overhead associated with a particular program without all the other extraneous references. The use of single user mode will depend on both the constraints of the kernel and the desired evaluation. Single user mode may also limit the choice of test programs. Some programs, such as SC in the SPEC benchmark suite, require specific interfaces which may not be available and so cannot be executed.

If the kernel is so slow that it cannot even be booted, it may be necessary to disregard some of the real-time interrupts that are stalling the system. The main interrupt of concern is the system call to the hardclock. The number of the hardclock calls which are performed can be scaled by using assembly code [10]. This allows a certain percentage of the interrupts to be ignored. This has by far the most significant impact on kernel speed, and should be sufficient to allow most programs to execute.

The speed factor also raises a question of accuracy. Any event that is based on an absolute timing mechanism (such as real time interrupts) will not be affected by instrumentation. That means that as an instrumented program executes, it sees a disproportionate number of these events during its execution. The hardclock scaling mentioned above will partially resolve this issue, but it has not been fully verified. Another accuracy factor is the number of context switches. If a system uses a maximum execution interval, the frequency of context switches seen by an instrumented test program will also be out of proportion. One measure used in [8] is to increase the maximum execution interval defined by the task scheduler.

3.3.5 Re-entrance

One of the most complex, and possibly significant, aspects of working with the kernel is its multi-threaded nature. System calls, interrupt service routines, and other overhead functions are all separate processes to be executed by the processor. They may be executed at any time during program or analysis execution. This causes a problem of guaranteeing the integrity of the analysis data. For example, during execution of the test program, the analysis routine is called. While the analysis routine is still processing that particular event, an interrupt occurs. The interrupt will supersede the analysis routine and the interrupt service routine will be executed. The service routine is part of the kernel, and is also instrumented. Therefore, as the service routine executes, it also generates events and calls to the analysis routines, *before the prior analysis routine call has completed.* Since all analysis routines access a common data structure, the actual state of the data becomes non-determinate and the evaluation results inaccurate. Consider an analysis routine which is interrupted in the middle of incrementing a counter. The counter is loaded and incremented, but has yet to be stored. The second execution of the analysis routine also increments the counter, so it loads, increments, and stores the data. The problem is, the value the second routine loaded was incorrect, since the first routine never had a chance to store the new value of the counter. When the first routine does return to execution, it then writes the value of the counter, which eliminates any changes to the counter that occurred during the interruption. Analysis functions must be designed explicitly to handle such concerns, called *re-entrant*, since they can effectively be "entered" multiple times without loss of integrity.

Further data thrashing is possible during a context switch. At a context switch, the current state of the processor is saved so that when that process returns to execution, it is started from the point where it was swapped out. This current status is usually represented by data such as the registers and allocation tables. In a threaded program, however, there may be data that is visible to all processes and not stored at the context switch. If this data is relevant to the state of a particular process, it must be explicitly defined as such. For instance, one process sets a variable in the global data. This data is carried over a context switch and is now visible to the next process, where it may or may not affect its execution. If the communication is intentional, care must be used so that a context switch performed in the act of setting the variable will not disrupt the execution. For this reason, the scope of data should be kept as local as possible, and any global data must be protected.

Re-entrance is normally achieved through synchronization. Each time a particular function is entered, it must determine if it is unique or if there are other instances of that function in mid execution. This is accomplished by a semaphore or other form of signal which is visible to all instances of every function. Such global data can be used to coordinate the activities of each function, the actual implementation depending on the desired effect. For the synchronization to be effective, it must be an atomic operation. The two acts of checking the semaphore and setting it if it is not already set cannot be interrupted, otherwise synchronization may be lost. For example, a process checks the signal and determines that it is the first instance of that analysis function. Before it can set the signal, however, an interrupt occurs and the function called again. This instance also checks the signal and determines that it is the first, conflicting with the legitimate first instance. Normal instructions do not provide this capability, as an interrupt may quite easily occur between testing and changing a variable. Instead, particular commands must be used, which will depend on the platform used.

The task of making analysis routines re-entrant is further complicated by the fact that the analysis routines are being executed within the kernel. There are many libraries of thread control and synchronization routines such as `pthreads.h`, `semaphore.h`, `signal.h`, and others, but these are mostly services provided *by* the kernel, not available within the kernel. To make the analysis routines fully re-entrant, it will be necessary to incorporate the same synchronization used within the kernel, which is not well documented.

In some cases the error introduced by data corruption is small enough that it can be tolerated. In other cases, contrived re-entrance can be incorporated with basic programming to insure some protection. For a detailed analysis of a multithreaded program such as the operating system, however, full re-entrance will be required. This problem has not been addressed before, and will require substantial investigation before it is adequately resolved.

3.3.6 Reference Stream Accuracy

The threaded nature of the operating system also raises accuracy concerns. Through testing, it has been determined that there is no duplication of kernel software similar to that used for shared libraries in single process simulation. This means that if the analysis routine in the test program makes a system call or instigates an interrupt, then the instrumented kernel service routine is executed. This in turn generates additional references for the simulation which would not have been

generated in the untraced version of the program. This is a significant concern, particularly if the execution of the operating system is to be analyzed in detail. Since all real-time interrupt routines are instrumented, they generate additional references as well since there is proportionately more interrupts per program execution time. To counter this, there must be an explicit mechanism to determine the cause of the operating system references and disregard the additional references — possibly something to incorporate as an aspect of the re-entrance mechanism.

3.3.7 Portability

The final area of concern is ATOM's portability. One criticism of many of the past methods was their lack of portability. Some are custom tools, and many were tied to a specific architecture or program. It is unfortunate that ATOM is no exception. ATOM has only been implemented for the DEC Alpha workstations and the operating system aspect can only be used with DEC OSF/1. The one advantage ATOM does have is its flexibility. Since it is a generic framework based on software, that framework can be reconstructed for other platforms or operating systems. The tools already created can then be used to compare results across systems. Because of this it is hoped that one day ATOM will be available for other systems, which is entirely possible.

4 Test Methodology

4.1 Cache Model

Fundamentally, a cache is simply a device used to store subsets of a large data pool for quick access. This type of structure may be found in a TLB [49], memory mapping tables [52], or within an instruction pipeline [27]. The most common form, and that which is modeled here, is a memory cache used to improve average memory access times by storing data mapped in from main memory. The design and execution of such caches have been rigorously studied, and are described in a variety of sources [22, 36, 52].

The goal for this research was to develop a flexible cache simulator that incorporates reference streams from multiple processes, including the operating system. This was built on the framework outlined in the previous section, using a common data structure in the kernel's address space to provide synchronization and store the cache state. The test program mapped this structure into the program's address space by accessing the `/dev/mem` facility, so all test programs must be executed as root (moot point in single user mode). To perform a single process simulation for comparison, the code was slightly modified so that the cache data was local to the test program, external communication and synchronization were no longer necessary. The code used is provided in appendix A, but a summary of the most significant characteristics is provided below.

The default ATOM tools only incorporate one test program and the operating system. By using the same technique, however, it is possible to extend a simulation to an arbitrary number of programs. Each program simply maps the same kernel data structure into its space via a pointer so each process now has access to the same common memory structure. In this way, simulations can be conducted with multiple test programs with the operating system.

For simplicity, the various analysis files were implemented as custom ATOM tools. This allowed the use of shared library functions such as math.h within the analysis functions, as well as simplified the act of instrumenting each test program. The tools defined for this research are:

`kexe` This specified the kernel instrumentation and analysis programs with the ATOM flags necessary to produce an executable version of the kernel.

`kdbg` Kdbg also specified the kernel instrumentation and analysis programs, but with the ATOM flags required to produce the debug version of the kernel used to map memory addresses.

user# The final tool was used for the test programs. The # symbol represents a digit, 1, 2, or 3, which identifies which test program is being instrumented. The only difference is the process identification number assigned.

The program captures both instruction and data references to be able to model both split and unified instruction and data caches. This is relatively simple for a RISC architecture; each instruction generates one instruction reference, and all data references are one of two possibilities, a data load or data store. Instrumenting every instruction generates too large a kernel to be executed on our system. Instead, instructions are instrumented within basic blocks in groups of 8 or less. This both decreases the size of the programs, and speeds their execution. The processing routine is passed the initial address and the number of instructions that follow to simplify processing. With this information, the addresses of each instruction can be recreated and processed. It is also possible to only instrument each basic block, but grouping instructions presents a problem. To simulate a unified cache, the interleaving of instruction and data references in the same stream is required. If instructions are instrumented in groups, the actual interleaving cannot be reconstructed. Data references could be out of place by as many references as the number of instructions grouped together. For this reason, instructions should be instrumented individually if possible. Using smaller blocks of instructions minimizes this error, and also allows another simplification in processing. If the groups of instructions are smaller than the cache block size, then only one reference need be processed for the entire group and the reference counter incremented by the group size. A small margin or error is introduced because of the assumption that instructions are aligned along blocks, but this will be minimal as block size increases. This was used in the simulator, limiting the minimum cache block size to 32 bytes given a 4 byte instruction.

Each reference is applied to its appropriate cache according to the cache's characteristics. The caches themselves are defined by 4 or 7 parameters, depending on cache type:

Type Either split, containing separate instruction and data caches (type = 1), or unified, having a single cache for both types of references (type = 0).

Cache Size The cache size in number of bytes. The size is specified as an area, so that the number of cache lines in a given cache is determined by:

$$\frac{\text{cache size}}{\text{block size} * \text{associativity}}$$

Cache size is specified independently for each section of a split cache, as are the last two parameters.

Block size The size in bytes of a cache block, which is the unit of transfer between the cache and memory.

Associativity The number of blocks per cache line.

For most simulations of this type, such parameters must be staticaly defined during compilation, which makes repeated tests with a range of parameters difficult. This is because the kernel cannot access file IO so simulation data cannot be loaded when the program starts. This program instead defines maximum parameters during compilation and memory is allocated for a worst case condition. When the operating system is started, the simulation also starts but with a flag so that all references are discarded. When the first test program is executed, it loads the desired cache parameters from a file and stores them into the cache structure, thereby allowing dynamic definition of simulation parameters. Once this is completed, reference capture is enabled and the simulation commences. This also speeds up the operating system when a simulation is not actually being performed, since after all test programs have completed the flag is restored and the simulation portion disabled.

Other cache characteristics are constant. These are programmed into the simulation and cannot be modified without code changes:

- The various threads encompassing the kernel are treated collectively as a single process.
- Caches are virtually addressed. A process identifier is associated with each cache block to identify its owning process, so cache flushes on context switches are not necessary. This neglects aliases, or multiple virtual addresses to the same physical location, but the effect of such shared data should be minimal given the test programs used. If multiple threads of a single process such as the kernel are to be considered, however, this cannot be ignored. Using virtual addresses drastically simplifies the simulation, since no translation to physical addresses is necessary, but it does have a drawback. The virtual addresses for a program will depend on the system executing it and how it has been mapped from memory. This mapping may be optimized for a particular memory system or the current execution environment, and so skew the results of a simulation of a different system on the same addresses. This must be accepted unless the virtual/physical mapping is also considered in the model, which is not a simple task.

Since the effect will be consistent across all programs and caches in the simulation, its impact is ignored.

- No prefetching (also called demand fetching) is incorporated into the simulation. This is not particularly realistic, since pre-fetching is a simple but powerful enhancement to cache performance, but for an initial test of the simulation capability, it becomes an unnecessary complication.
- All references are assumed to be the same size, accessing a single byte. This is acceptable assuming that any words addressed do not cross cache block boundaries.
- Mapping of addresses to cache lines is by a simple masking of the low order address bits. This is the most simple and common form, although other hashing algorithms are possible.
- An allocate on write policy is used, so data writes are treated the same as reads. This is generally the most pessimistic write policy, as opposed to its opposite, no fetch on write, in which a data write miss is ignored by the cache and sent directly to memory [29]. Write back versus write through considerations are ignored, as the model does not consider traffic to main memory.
- Set associative caches use a least recently used (LRU) replacement algorithm.

Cache performance is recorded as reference and miss totals for each type of reference. Totals are generated separately for each process for each cache. Values are reported at the end of the simulation; for multiple processes at the end of each process. Process overwrite data is also captured, in the form of the total number of overwrites by each process over each of the other processes. This is accumulated by incrementing a particular counter identifying the previous and present owning process for each cache block overwritten. Cache performance information for the operating system is only captured during the execution of test programs. References before or after the program are ignored.

One concern was that in a multiprocess environment, execution is non-deterministic. Because of this, multiple executions cannot be used to evaluate multiple caches, as there will be differences between each execution. To counter this, multiple caches with varying characteristics are simulated during a single execution. This way, cache performance can be compared across equivalent loading. It does slow down execution, but accomplishes more with one run.

Another concern was the threaded characteristics of the operating system analysis, some form of re-entrance was required. To address this, a flag is set upon entry to the ATOM analysis routines. The flag is a global variable visible to all of the executing processes, so can be used for synchronization. If an analysis routine encounters the flag already set on entry, it immediately exits, maintaining data integrity. By assuming that the reference which called the analysis routine was in some way instigated by another analysis routine, this also prevents interrupts generated by the analysis routine from contributing to the simulation reference stream. It does cause any other interrupts which occur during analysis processing to be neglected as well. While this may seem like a disadvantage, such real-time interrupts are normally skewed by the slowed processing, so neglecting a portion of them is actually beneficial. This implementation is not ideal, because the flag is not set or cleared as an atomic operation. The majority of signaling and synchronization protocols available in programming are actually services provided by the kernel, and therefore not available to code that is executing within the kernel. If an interrupt occurs in the process of checking or setting the flag, the execution is undetermined. This was particularly a problem during context switches, so another mechanism was added. Not only do the analysis routines check the signaling flag, but they also check to see if a context switch has occurred. If a context switch has occurred, the flag is automatically reset. This is obviously a very improvised strategy and has much room for improvement, but it was effective in regulating the reference stream enough to allow reasonably accurate simulations.

Other aspects of the code were dictated by the use of ATOM. As mentioned in the previous section, all processing was kept to a minimum. Loops were used sparingly, and no function calls beyond the original analysis routine were used. This is not particularly good software engineering practice, but necessary. The hardclock scaling mentioned was also incorporated, with a 90% reduction in the number of hardclock calls. Even with these measures, the instrumented operating system was slow enough that it was also necessary to perform all simulations in single user mode. Multiple processes could still be used by executing them in background mode.

The program developed is a very comprehensive and flexible simulator with a great deal of potential, but it does have some problems discovered in hindsight that should be addressed in future work.

- Program size is still a concern; more memory is definitely needed to reduce paging for more accurate simulations. Increasing memory should also improve execution times.

- Program speed is also still a concern. Ideally, the scheduler should have been modified so that instrumented programs use a longer maximum execution interval to accommodate their decreased speed as done in [8].
- The block replacement data showing process overwrites is not distinguished by reference types. This is an oversight and limits the potential usefulness of the data, as it is impossible to determine the contribution of each type of reference to the amount of interference.
- Using virtual addressing is simplistic and raises other issues. Physical based addressing should be used if possible.
- The impact of the existing memory system and architecture are not considered, simply assumed to be consistent and neglected.
- The methods used to correct timing problems, such as scaling hardclock interrupts and ignoring interrupts during analysis, are not verified. An extensive analysis should be conducted to demonstrate or refute their effectiveness.
- The synchronization used is very fragile. Ideally the synchronization method used within the kernel should be studied and incorporated so that the analysis code is truly re-entrant. This is particularly necessary for more reliable analysis of threaded programs.

Even with these potential problem areas, however, the program was capable of performing most of the desired simulations, and provided an adequate validation of the multi-process capability of ATOM.

4.2 Verification

To have any confidence in the results of a simulation, the simulator must first be verified to ensure that it does indeed produce accurate results. The developmental nature of this project precluded a direct comparison with other equivalent work. Default tools are provided with ATOM which can incorporate the operating system, but do not have the flexibility to verify the range of cache types that will be simulated. Other tools are not readily available to generate comparable simulations. Instead, a multi step approach was used to demonstrate the program's correctness.

The first concern was the ability of the program to accurately capture the address traces. This was accomplished by writing a second ATOM based application that simply captured traces

without performing any other processing. The references it captured were compared to those captured by the simulator, which were identical. The second ATOM tool was simple enough that it could be verified by inspection, so if it does not capture the address traces correctly then any flaw is within the ATOM framework and cannot be addressed here.

The next aspect to be verified was the processing of the reference stream. The program was slightly modified so that as each reference was processed, it was also stored to file. A trace file was generated for the following four benchmarks:

- Compress
- Ear
- Espresso
- SC

for the three caches shown:

- Unified 8192 byte 2 way associative cache with 64 byte blocks
- Split 2048 byte fully associative caches with 32 byte blocks
- Split 4096 byte direct mapped caches with 32 byte blocks

The trace file was then used as input to the DineroIII cache simulator to test the cache processing. DineroIII and simulation results were identical for all 12 cases.

A further test was used to ensure the simulation program executed correctly. The results of single process simulations were compared to the results of benchmark cache analysis in other papers [25, 45]. The cache performance was roughly the same in that the same general behavior patterns were present, however there were some differences. This is primarily due to differences in the inputs used; in some cases alternate or combinations of inputs different than those used here were simulated by the previous research. Their results were also generated from optimized code which disregarded shared library references. For our tests, code was not optimized and all references are captured, so the difference is to be expected.

The final concern regarding the simulator was its repeatability. Given the threaded environment, results could vary within a single execution. Given the non-deterministic environment, results could also vary over multiple executions so an experiment was conducted to determine the extent of

the possible variation. The same three caches mentioned above were simulated for Compress, Ear, and Espresso 5 times each in succession. Each simulation modeled ten identical caches. The first results showed that not only did performance vary, but so did the reference load. Each successive execution of the same program after the initial execution had a reduced number of references from the kernel. Upon reflection, we realized that this was due to the overhead required for the first execution of loading the program into memory. All following executions had reduced operating system overhead since the test program was already in memory, as can be seen in Figure 2.

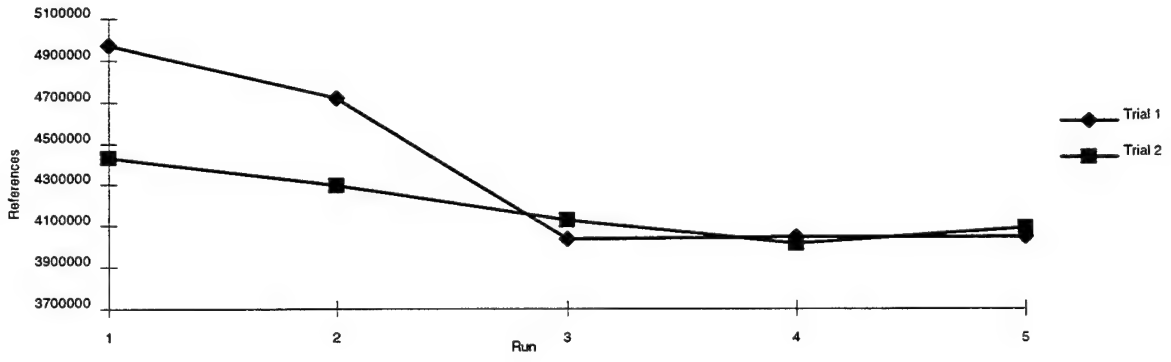

Figure 2: Operating System Instruction Fetches Over Repeated Program Execution

To eliminate this factor, the tests were repeated without having each program executed sequentially. The variation was reduced, but not eliminated. For complete accuracy, the system was rebooted between all later simulations. The second set of results highlighted another problem. In the output file, the operating system references varied even through the process of recording the results to file. Figure 3 shows the number of kernel instruction references for ten identical caches from the same simulation. The increasing number of references for the later caches suggests the point made in the previous section, that in the operating system environment, ATOM does not correctly distinguish between calls to common code made from the test and analysis sections of the program.

The variation within a single simulation was also due to the threaded nature of the analysis, so the pseudo re-entrance measures discussed above were then incorporated into the program. They eliminated the majority of the operating system references generated by the simulation routines, as well as prevented most of the data thrashing. The simulations were again repeated, although only for the Espresso benchmark and only for 2 split caches, fully associative and direct mapped. These

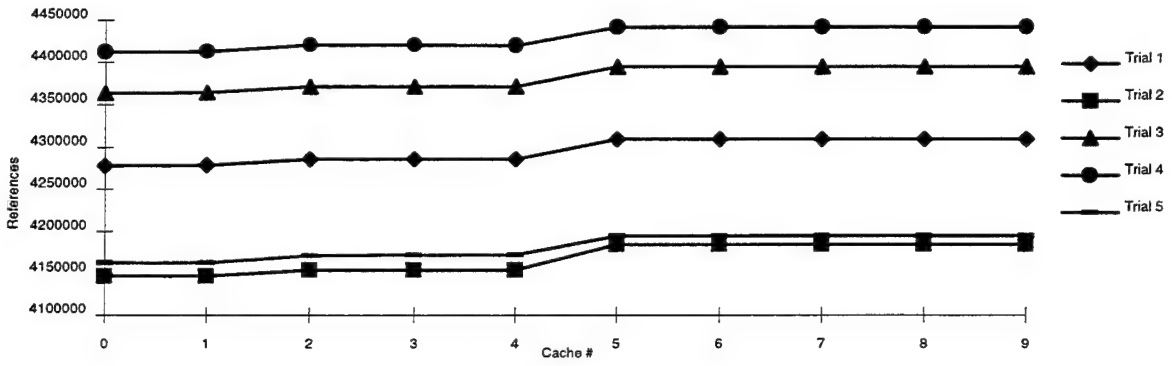

Figure 3: Operating System Instruction Fetches Within Same Program Execution

results showed no variation at all within a single execution, and only a minor variation of .01 to .1 in the cache miss rates between different executions. Prior to these measures being taken, the worst variation was substantially less than was expected, however using a single user mode for execution limits the number of extraneous processes and greatly reduces the non-determinism of execution. With the additional precautions, we are confident in the accuracy of the simulation results.

4.3 Simulations

4.3.1 Platform Information

The described tests were performed on a DEC Alpha 3000 model 300, a RISC based AXP architecture. The root partition had to be expanded to 85MB to accommodate the larger kernels used, which could contain up to a 48MB test kernel in addition to the normal root residents. The swap space was originally 195MB which proved to be insufficient to instrument large programs. A second local disk was added increasing the swap space to 323MB. The usr partition was 694MB which was generally adequate although more space was useful at some points. The added disk included a 1090MB scratch directory which proved to be invaluable in storing results, traces, kernels, and other files. The critical factor was memory. The system only had 64MB of main memory, so during simulations only about 15MB of memory was available for test programs. For future efforts, the memory must be increased to improve simulation performance and accuracy.

The operating system used was DEC OSF/1 version 3.2A Unix kernel. Newer versions are available however this version was sufficient for these tests. The ATOM tool used was version 2.20. It is also being continuously updated; research was begun with version 2.13, although the system was

upgraded to version 2.20 before simulations were performed. Each new version of ATOM usually addresses shortcomings of past versions, particularly in terms of intrusiveness, and refines the newer capabilities, such as instrumenting the kernel, so the most current version available should be used for future work. The test programs used are from the SPEC 92 benchmark suite. These programs tend to focus on technical, as opposed to commercial, applications. They are more computation intensive than other potential test programs, but are also readily available and a standard test tool.

4.3.2 Test Parameters

Simulations were performed capturing cache miss rates for program execution alone, programs with the operating system, and multiple programs executed concurrently. The four benchmarks used for these simulations were [74]:

Compress The compress benchmark is the same program as the Unix compress utility. It is a CPU intensive integer benchmark which compresses an input file using the Lempel-Ziv data compression algorithm. It has a greater IO content than the other benchmarks, so is more sensitive to the system and execution environment. Due to its nature, the program has a repetitive instruction reference stream with a drastically less localized data reference stream. A 1MB input file `in` was used with the following command line:

```
#compress -f -c in > /dev/null
```

which causes the utility to route the compressed data to stdout instead of back to the original file, where it is discarded. This was done so that the execution of the benchmark did not affect the input program, which was useful during repeated executions. As part of the benchmark suite, the test calls for multiple iterations of compress, but for our tests only a single execution is performed to reduce simulation time. The goal of this research is not to benchmark the system used, so the full tests were not required.

GCC GCC is the GNU C compiler, and is the most complex benchmark used. As a compiler, the parsing, organization, and optimization performed produce a highly irregular reference stream. Some IO is performed, as well as a variety of other system calls, and the execution depends heavily on the system used. The compiler was executed by:

```
#gcc -O -quiet stmt.i -o stmt
```

which caused it to optimize the source code and suppress any output. Again, the benchmark suite called for compilation of multiple programs, however only the single input `stmt.i` was used for simplicity. One note regarding the instrumentation of gcc, it does require certain ATOM flags the other three benchmarks do not. The ATOM command line to be used with gcc is:

```
%atom gcc.rr -tool user1 -heapbase 50000 -32addr
```

These are required for ATOM to correctly instrument gcc, as the compiler uses a wider range of the address space and a larger heap segment of memory.

Espresso Espresso is a tool for generating and optimizating Programmable Logic Arrays. Its primary task is minimizing Boolean functions, so also has a repetitive instruction stream with a more localized data stream than compress. It uses very few operating system services, and is a small program (before tracing), so normally requires little paging. The benchmark was used with the `tial.in` input file with suppressed output as shown below:

```
#espresso tial.in > /dev/null
```

As the other programs, the actual benchmark entails multiple input files, but only this one was used for testing.

Alvinn Alvinn stands for Autonomous Land Vehicle in a Neural Network, and represents a neural network control system capable of taking data from a video camera and laser range finder and generating control data for an automated vehicle. The benchmark is a single precision floating point program which trains the network through backpropagation over 200 input epochs. It performs minimal IO, although does use the floating point unit extensively. It is repetitive, although with a much more complex structure than Compress. The command line used was simply:

```
#backprop > /dev/null
```

which activates the training model with the input files `h_o_w.txt`, `i_h_w.txt`, `in_pats.txt`, and `out_pats.txt` residing in the test directory. The results of the training for each epoch are the only output, which is discarded.

Each simulation was performed as described in the previous sections using an input file of 40 caches of various configurations. Table 1 assigns a number to each cache which is used for later identification, and shows the different characteristics of each. Only lower associativities are used to minimize the amount of looping in processing. Other characteristics are arbitrary selections over a general range, with a limit of 512 lines per cache to minimize storage. The results of these simulations are discussed in the next section.

		Unified or Instruction			Data		
ID	Type	Cache Size	Block Size	Assoc	Cache Size	Block Size	Assoc
0	0	8,192	64	2	NA	NA	NA
1	0	16,384	64	2	NA	NA	NA
2	0	32,768	64	2	NA	NA	NA
3	0	65,536	64	2	NA	NA	NA
4	1	4,096	32	1	4,096	32	1
5	1	4,096	32	2	4,096	32	2
6	1	4,096	32	4	4,096	32	4
7	1	4,096	64	1	4,096	64	1
8	1	4,096	64	2	4,096	64	2
9	1	4,096	64	4	4,096	64	4
10	1	4,096	128	1	4,096	128	1
11	1	4,096	128	2	4,096	128	2
12	1	4,096	128	4	4,096	128	4
13	1	8,192	32	1	8,192	32	1
14	1	8,192	32	2	8,192	32	2
15	1	8,192	32	4	8,192	32	4
16	1	8,192	64	1	8,192	64	1
17	1	8,192	64	2	8,192	64	2
18	1	8,192	64	4	8,192	64	4
19	1	8,192	128	1	8,192	128	1
20	1	8,192	128	2	8,192	128	2
21	1	8,192	128	4	8,192	128	4
22	1	16,384	32	1	16,384	32	1
23	1	16,384	32	2	16,384	32	2
24	1	16,384	32	4	16,384	32	4
25	1	16,384	64	1	16,384	64	1
26	1	16,384	64	2	16,384	64	2
27	1	16,384	64	4	16,384	64	4
28	1	16,384	128	1	16,384	128	1
29	1	16,384	128	2	16,384	128	2
30	1	16,384	128	4	16,384	128	4
31	1	32,768	64	1	32,768	64	1
32	1	32,768	64	2	32,768	64	2
33	1	32,768	64	4	32,768	64	4
34	1	32,768	128	1	32,768	128	1
35	1	32,768	128	2	32,768	128	2
36	1	32,768	128	4	32,768	128	4
37	1	32,768	256	1	32,768	256	1
38	1	32,768	256	2	32,768	256	2
39	1	32,768	256	4	32,768	256	4

Table 1: Simulated Cache Parameters

5 Simulation Results

Simulations of caches with varying types, cache sizes, associativities, and block sizes as described in Table 1, were performed with the 4 benchmarks. The data generated by the simulations has been analyzed by focusing on various aspects of the cache behavior. These are the change in cache workload, the change in cache performance for a specific process, the interference generated between the processes, and the net change in cache performance over all processes. Other areas of possible exploration include studying performance differences between data reads and writes, and a detailed characterization of the operating system performance. In some instances only a portion of the available data is shown in figures. Tables of all results are provided in appendix B.

5.1 Cache Workload

Before looking at the cache performance, it is important to understand how introducing the operating system and additional processes affect the memory reference stream. The first set of simulations establish a baseline by recording the cache's performance for each benchmark alone. The frequency of each type of reference is presented in Table 2.

Benchmark	Instruction Fetches	Data Reads	Data Writes	Total Data	Total References
Compress	87,045,943	22,412,017	8,521,660	30,933,677	117,979,620
GCC	160,240,141	50,197,329	19,074,844	69,272,173	229,512,314
Espresso	977,787,923	225,779,346	59,867,420	285,646,766	1,263,434,689
Alvinn	5,233,222,111	1,415,013,652	487,428,474	1,902,442,126	7,135,664,237

Table 2: Benchmark References

The second set of simulations used the same benchmarks, but included the operating system. The frequency of each type of reference is shown in Table 3 for each process. There is some variation in the number of references for each benchmark due to execution differences, but it is minimal. Hello World was used for some of the basic program testing, and is included as a curiosity. For the other benchmarks, the operating system overhead was generally small, less than 15% of the total number of references. For a small program such as Hello World, however, the operating system overhead becomes the dominant source of memory references, totally overshadowing the program.

The amount of overhead introduced by the operating system is smaller than expected. This is because the tests were performed in single user mode, and a majority of the operating system routines were not being executed. In this context, processes such as network and printer controllers,

and the variety of other background system processes are considered to be part of the 'operating system'. One test using ps in multi-user mode showed over 40 different processes being executed, only one of which was actually a user program. For these system processes to be included, they must also be instrumented. During the simulations performed, the operating system references are generally just the overhead required by the test programs.

Benchmark	Instruction Fetches	Data Reads	Data Writes	Total Data	Total References
Hello World	1,247	207	135	342	1,589
OS	337491	84,403	51,332	135,735	473,226
Total	338,738	84,610	51,467	136,077	474,815
Compress	87,045,969	22,412,010	8,521,661	30,933,671	117,979,640
OS	5,567,602	1,518,924	802,242	2,321,166	7,888,768
Total	92,613,571	23,930,934	9,323,903	33,254,837	125,868,408
GCC	160,240,175	50,197,333	19,074,845	69,272,178	229,512,353
OS	18,705,569	5,130,601	2,613,506	7,744,107	26,449,676
Total	178,945,744	55,327,934	21,688,351	77,016,285	255,962,029
Espresso	977,787,899	225,779,331	59,867,421	285,646,752	1,263,434,651
OS	29,093,428	9,107,479	3,585,537	12,693,016	41,786,444
Total	1,006,881,327	234,886,810	63,452,958	298,339,768	1,305,221,095
Alvinn	5,233,222,045	1,415,013,630	487,428,474	1,902,442,104	7,135,664,149
OS	197,365,478	60,413,211	25,986,851	86,400,062	283,765,540
Total	5,430,587,523	1,475,426,841	513,415,325	1,988,842,166	7,419,429,689

Table 3: Benchmark with Operating System References

The operating system overhead will vary depending on the nature of the program, but for these benchmarks it remains fairly consistent. The percent of the total references which are generated by the kernel is shown in Figure 4, which ranges between 2.89 to 12.05 percent. This can also be viewed as the percent increase in number of references as seen in Figure 5, which has a similar range. For the benchmarks used, the program references still dominate. The benchmarks which require minimal resources and I/O (Espresso and Alvinn) are the least affected by the addition of the operating system. Compress is also fairly simple, but requires a larger amount of I/O, hence its greater overhead. A complex program such as the GCC compiler is affected the most. The amount of overhead found in these results is less than that found in past studies [1, 2]. Agarwal found the operating system could increase the number of instructions by 5-75%, but this is also for an older, CISC, architecture. Both studies did show that complex programs, such as compilers, are the most affected.

Figure 6 shows the relative distribution of each reference type within the workload for both the program and its operating system overhead. Both the program and operating system references have about the same distribution, with roughly 70% instruction fetches. This is consistent with

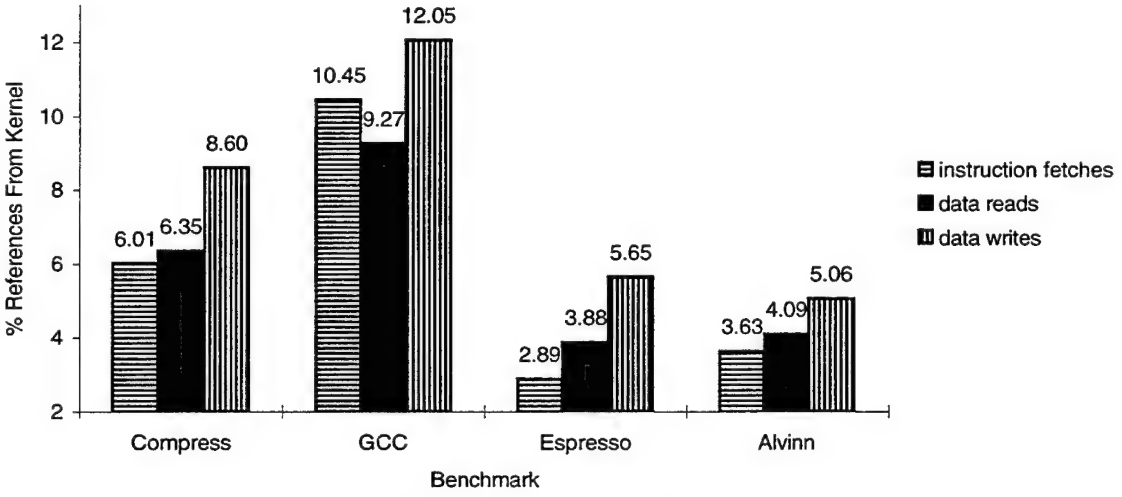

Figure 4: Percent of Total References From Operating System

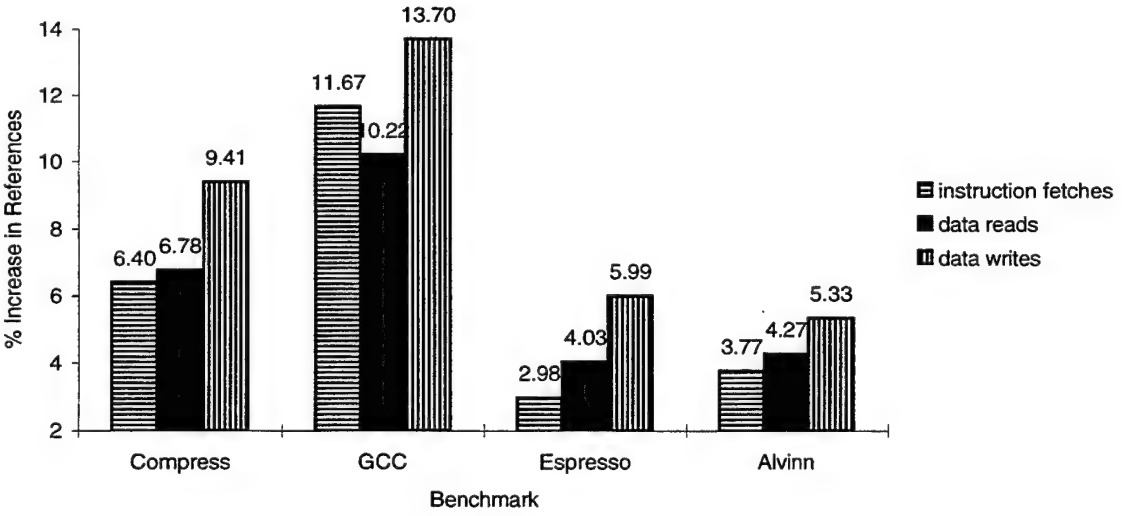

Figure 5: Percent Increase in Number of References by Including Operating System

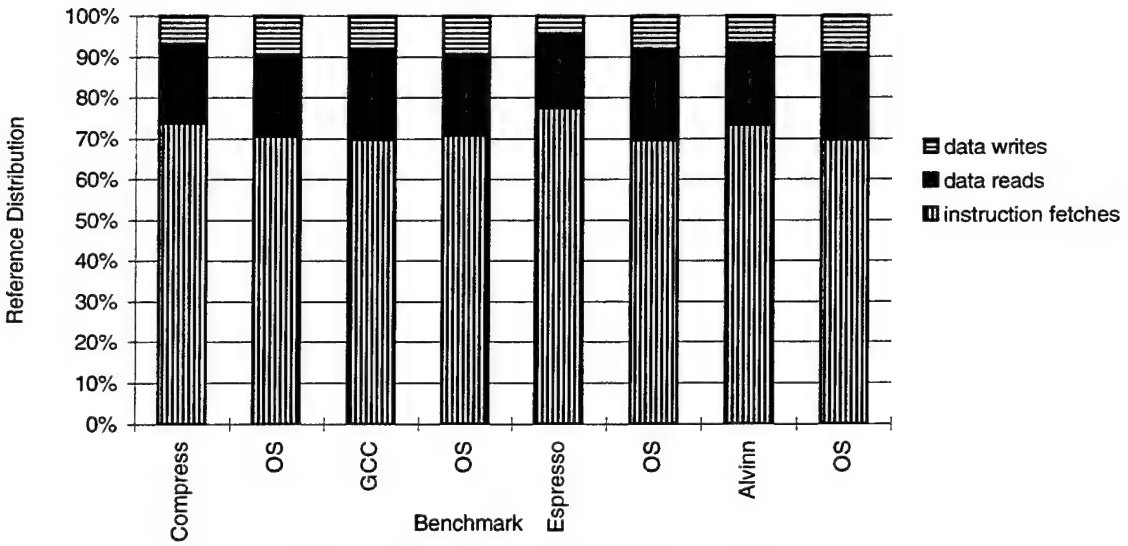

Figure 6: Distribution of Reference Types

[8]. The small proportion of data writes explains the seemingly larger change seen in the previous two figures — there are relatively fewer data writes so a smaller change generates a larger percent difference.

The final set of simulations was performed executing two benchmarks concurrently, capturing references from each and the operating system. Results were logged after each test program completed. The first report contains the information of interest, the cache performance with two competing user programs. The second report includes the period of time after the first process had completed, so only a single user process was executing during part of its tracing period. Since this analysis focuses on the effects of multiple processes, the second report has been discarded. For this reason, the data shown in Table 4 omits a portion of the execution of the longer process in each case. Any future references to these simulations also refer specifically to the cache performance at the end of the first program.

One fact that is not visible from this table is that when both programs have completed, the cumulative operating system overhead (measured in number of references) is greater than the sum of the overhead for each program individually, as shown in Table 5. If the number of operating system references generated when the benchmarks are executed separately are added (the first column), this value is less than the number of operating system references generated when the same two benchmarks are executed concurrently (the second column). This highlights the increased operating system activity required to switch between multiple processes, roughly a 20–40% increase.

Benchmarks	Instruction Fetches	Data Reads	Data Writes	Total Data	Total References
Compress	87,045,885	22,411,994	8,521,651	30,933,645	117,979,530
GCC	68,021,687	21,218,807	8,094,452	29,313,259	97,334,946
OS	28,102,411	7,468,658	4,160,003	11,628,661	39,731,072
Total	183,169,983	51,099,459	20,776,106	71,875,565	255,045,548
Compress	87,045,885	22,411,994	8,521,651	30,933,645	117,979,530
Espresso	99,475,944	24,280,822	4,659,787	28,940,609	128,416,553
OS	15,541,809	4,310,868	2,247,254	6,558,122	22,099,931
Total	202,063,638	51,003,684	15,428,692	66,432,376	268,496,014
GCC	160,240,175	50,197,333	19,074,845	69,272,178	229,512,353
Espresso	224,015,827	51,131,704	12,097,918	63,229,622	287,245,449
OS	39,004,710	10,758,087	5,592,574	16,350,661	55,355,371
Total	423,260,712	112,087,124	36,765,337	148,852,461	572,113,173

Table 4: Concurrent Benchmarks with Operating System References

Benchmarks	Sum of Individual Overheads	Concurrent Overhead
Compress/GCC	34,338,444	47,433,154
Compress/Espresso	49,675,212	59,365,363
GCC/Espresso	68,236,120	89,030,467

Table 5: System Overhead Comparison

A problem arose when certain programs (or combinations of programs) were traced, generating the `trap: invalid memory access` error mentioned previously. It is somehow related to the size or length of the test programs. Benchmarks such as Xlisp (9,561,089,165 references) and Ear (17,375,158,291 references) would crash the platform if simulated with the operating system. Similarly, executing any of the three smaller benchmarks concurrently with Alvinn would crash the system, as well as any three programs in combination. While this problem limited the scope of the simulations, correcting it was beyond the purview of this research.

5.2 Impact on Process Performance

The simplest way to visualize the impact of the operating system and additional processes is to measure their effect on the cache performance for a particular program's reference stream. Figures 7 through 14 show the cache miss rates for benchmark references only, for each of the 4 benchmarks. The baseline is the result from the single process cache simulation. The other sets of results are essentially the same reference stream but with transient misses. Any performance changes are due strictly to these transient effects.

The single process results exhibit normal cache behavior. As expected, increasing cache size decreases miss rate. A larger cache can contain more, if not all, of a programs working set,

thus reducing capacity misses. Also, a larger cache will have fewer locations assigned to each line, potentially reducing conflict misses. Increasing associativity also decreases miss rates, although with diminishing returns; the improvement from A=2 to A=4 is less than the improvement from A=1 to A=2. Associativity can reduce conflict misses by allowing a line to maintain more than one block at a time, but the benefits are limited by the number of references to any one line. Since the caches use a constant area, increasing the associativity decreases the number of possible indices, thus increasing the stress on a single index. For this reason, in some instances increasing associativity can *increase* the miss rate (e.g. Alvinn). Increasing the block size increases the amount of memory fetched on each miss. This is generally beneficial for instruction references which exhibit spatial locality, but the reverse may be true for data references. Depending on the benchmark, data miss rates can either increase (e.g. Compress) or decrease (e.g. Espresso) as block size increases, but this trend is also related to associativity and other factors. Increasing block size also decreases the number of cache indices, so again the load on each line is increased potentially negating any benefits. These results are comparable to those found in [25, 45, 56].

Comparing the single process results with the other simulations, these trends are not generally affected. In most cases, the results follow the same patterns but with a noticeable increase in cache miss rates. The amount of increase may vary by cache or remain relatively constant, depending on the characteristics of the particular benchmark being considered. This increase is the error in assuming that cache behavior can be defined by a single process simulation, and shows the difference between a single program's cache performance when it is considered alone versus when it is considered in a multiprocess simulation. As can be seen, the impact of the operating system is much smaller than that of an additional process. This is logical, considering the operating system normally executes for shorter durations as it services system calls and interrupts. The impact of additional processes is generally most pronounced in those caches that already exhibit poor performance, although this does depend on the benchmark.

It is also interesting to consider the distribution of misses. Figures 7 through 13 show the percent of misses that were from instruction references. It is interesting to note that although instructions make up the majority of references, they are usually in the minority of misses — as expected due to their increased locality. For programs such as Compress or Alvinn with a great deal of spatial locality in their instructions but not data, the loss of locality due to transient interference is visible in the increased proportion of instruction misses found in the simulations which included

the operating system and additional processes. Other programs such as Espresso may be affected either way, although data misses still predominate. A more complex program such as GCC has much less locality in its reference stream, as can be seen by the fact that instructions account for as much as 65% of its misses. Hence when the additional processes are considered, it is possible for data cache hit rates to be affected more and the ratio to go down.

5.3 Process Interference

Another way to visualize the impact of the additional references is to analyze the proportion of intrinsic versus extrinsic interference seen by the various test programs. The percentage of misses attributed to intrinsic interference can be approximated by the percent of misses where the reference overwrote a block containing information from the same program. The alternative is for the reference to miss and overwrite another program's data, highlighting extrinsic interference. A certain number of references will miss and overwrite invalid data at start up, but these are finite (based on cache size), and will not significantly affect the percentage. The self overwrite percentage is shown for each cache for the 4 benchmarks in Figures 19 through 22. When a block is overwritten no test is performed to see if the evicted data is live, nor is there a check of the new data to determine if it has been accessed before, so these figures are not exactly intrinsic interference, but should be comparable.

The most basic simulation with a single benchmark as input will have 100% of its misses due to internal considerations, by definition. When the operating system is added, roughly 10–20% of the misses are external overwrites, due to the impact of the OS references. Adding an additional process to the simulation increases the external impact to 40–70%, depending on the cache and particular program. It is unfortunate that it was not possible to perform simulations with a greater multitasking level so that a trend might be visible.

Smaller caches are affected more by extrinsic interference as expected, as are caches with lower associativities. As each process is executed, its references are loaded into the cache. A smaller cache may be totally overwritten by the new data, while a larger cache may be able to retain a portion of the previous program's working set. Program characteristics such as the amount of system overhead, as well as working set size and fluctuation, affect the amount of interference, but are more difficult to quantify without an extensive trace analysis.

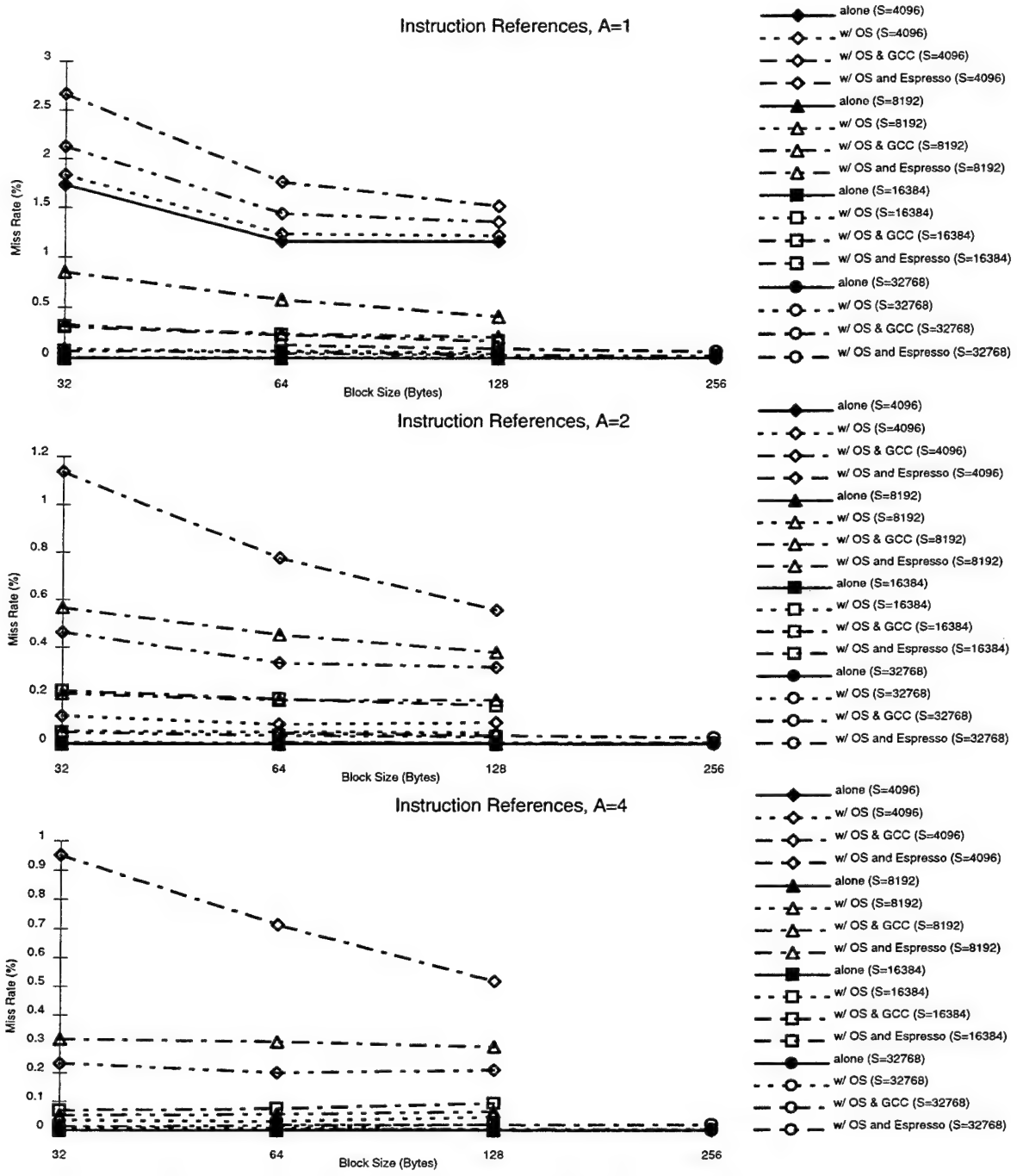

Figure 7: Process Instruction Reference Miss Rates For Compress

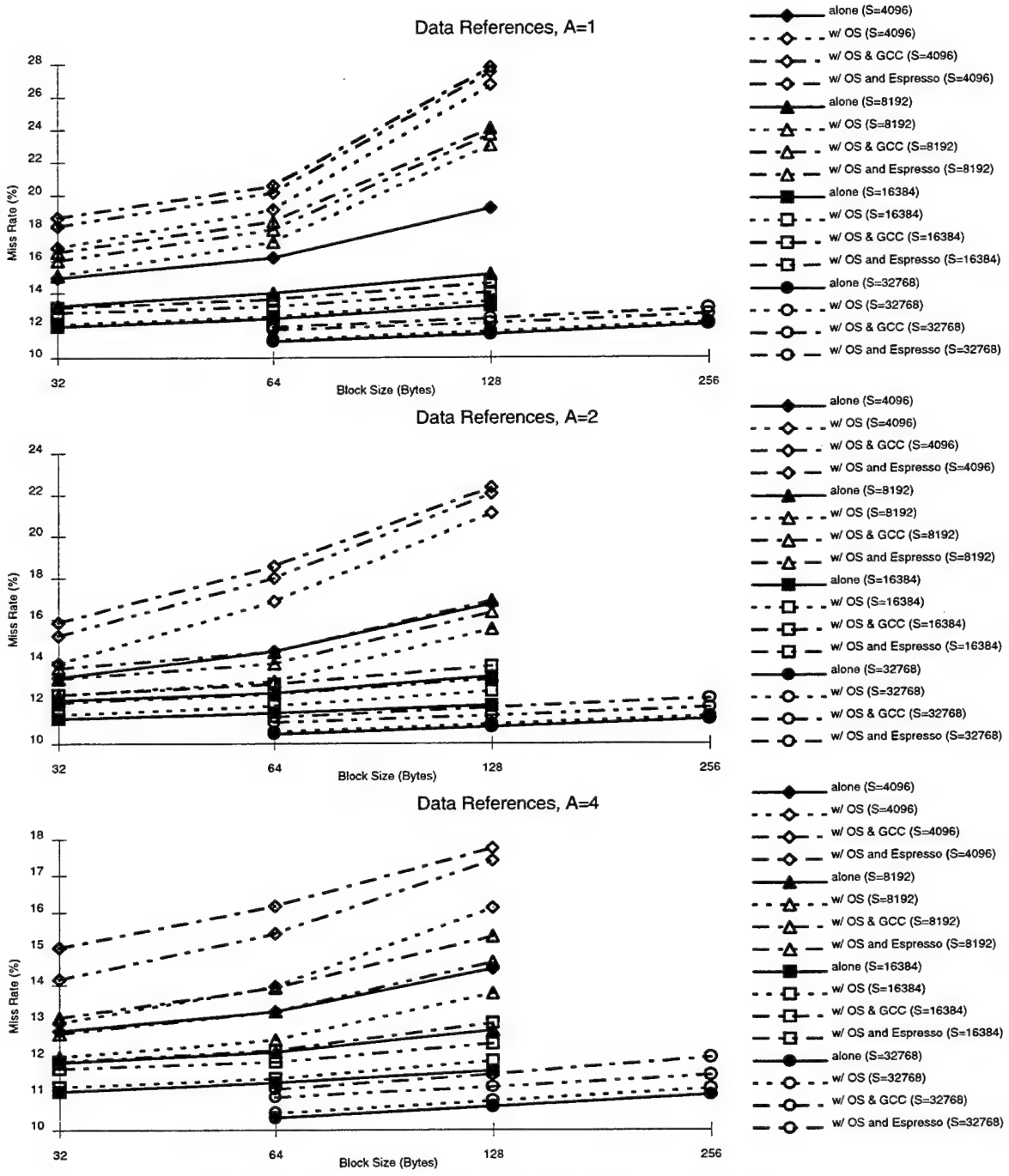

Figure 8: Process Data Reference Miss Rates For Compress

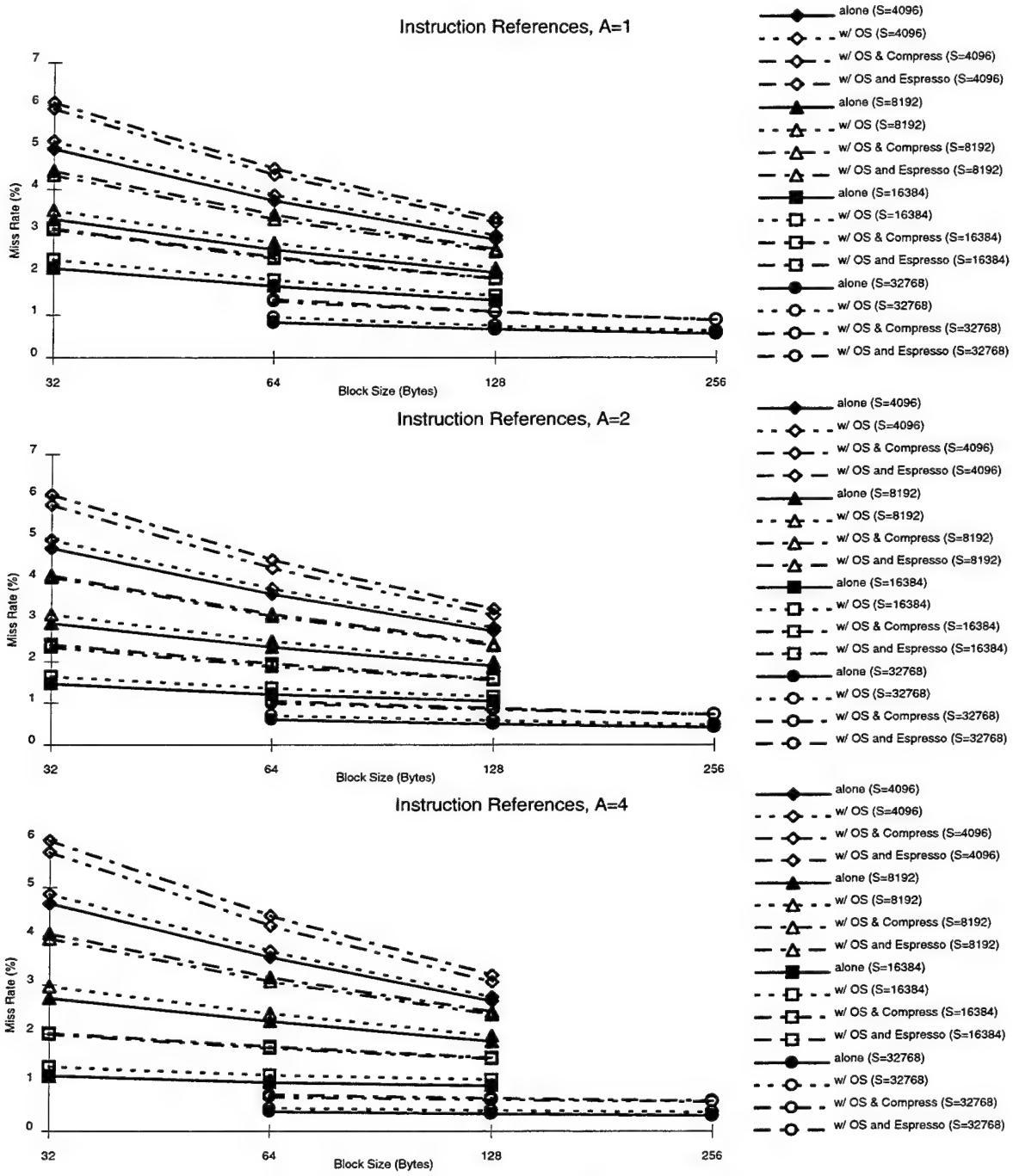

Figure 9: Process Instruction Reference Miss Rates For GCC

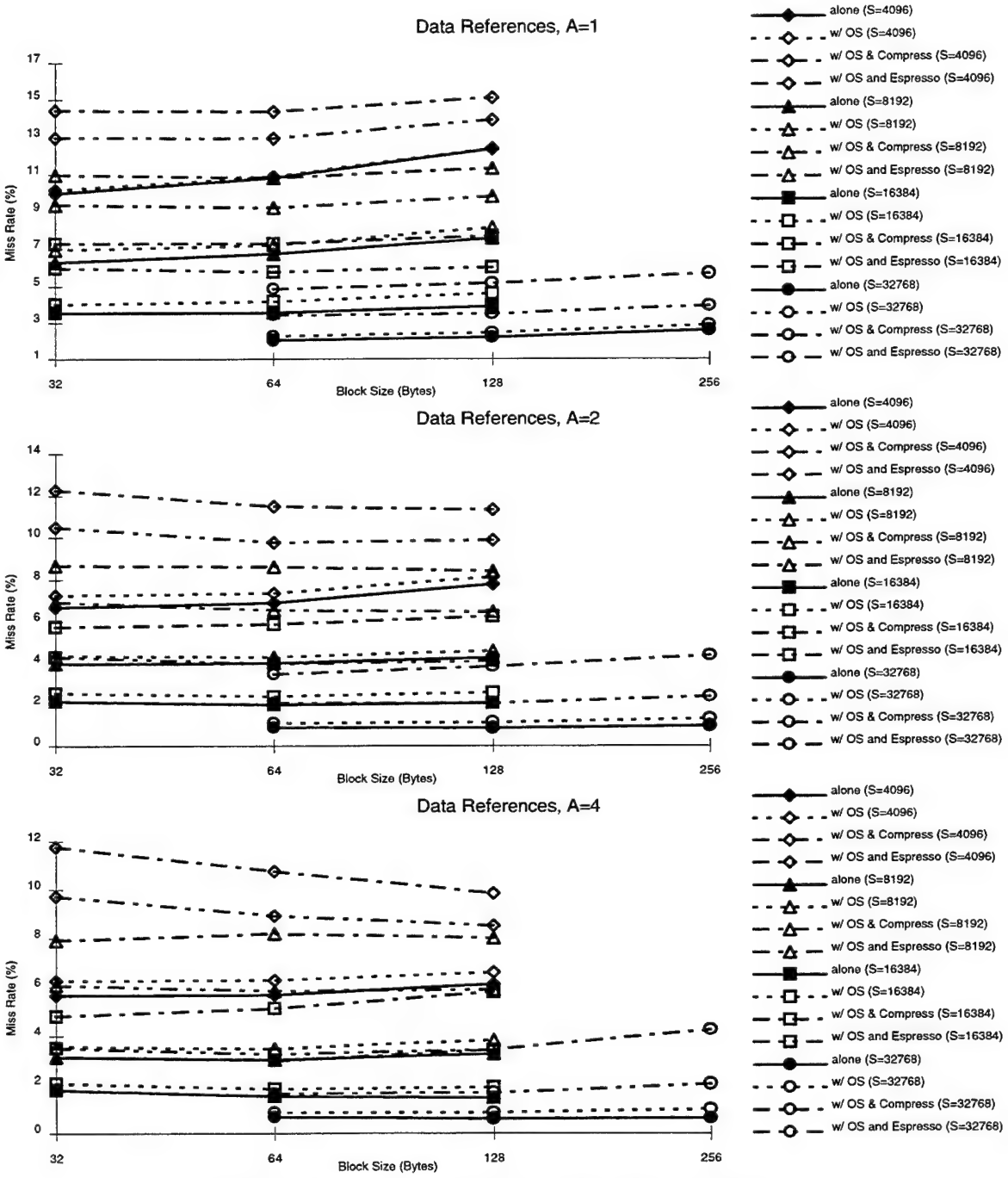

Figure 10: Process Data Reference Miss Rates For GCC

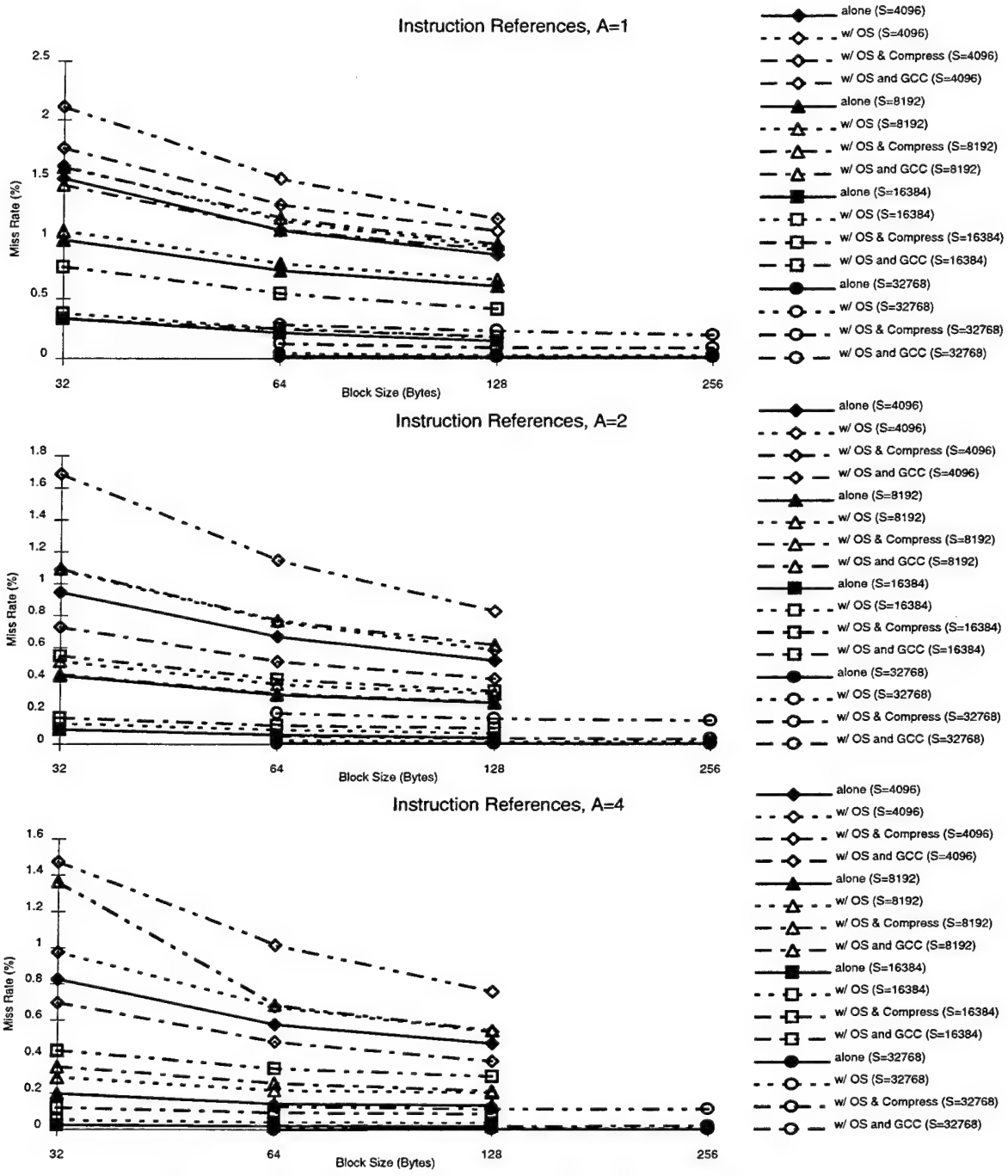

Figure 11: Process Instruction Reference Miss Rates For Espresso

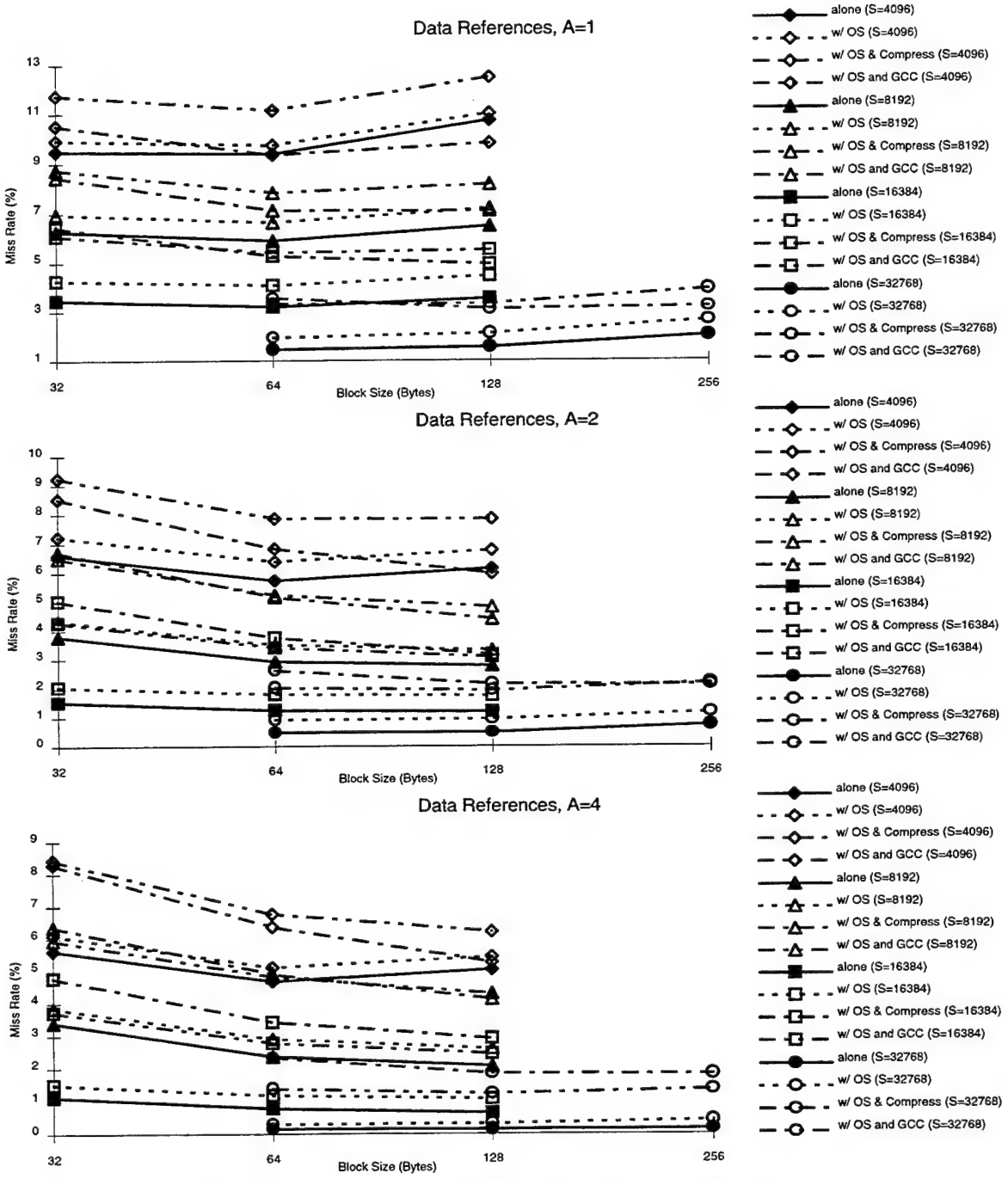

Figure 12: Process Data Reference Miss Rates For Espresso

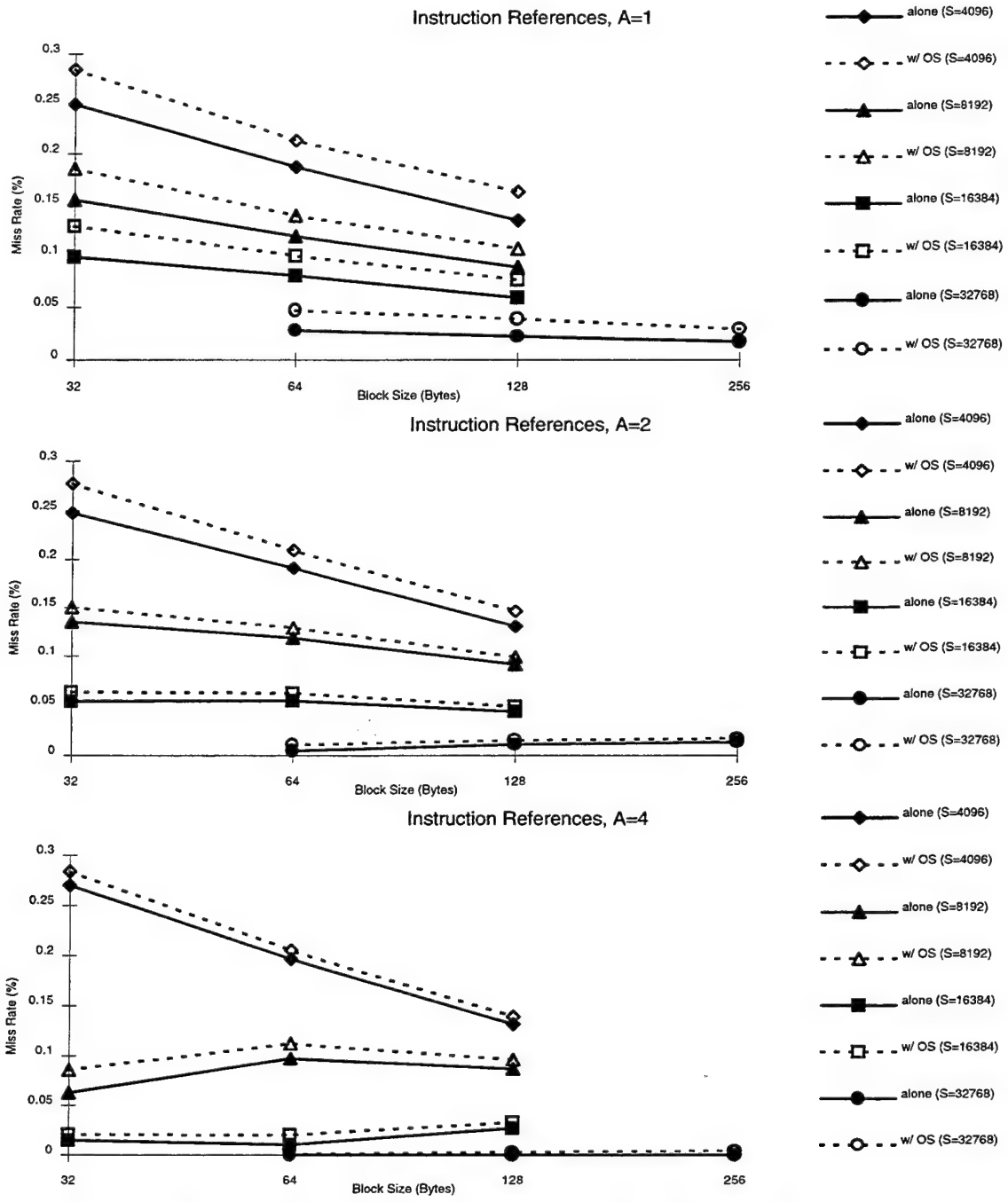

Figure 13: Process Instruction Reference Miss Rates For Alvinn

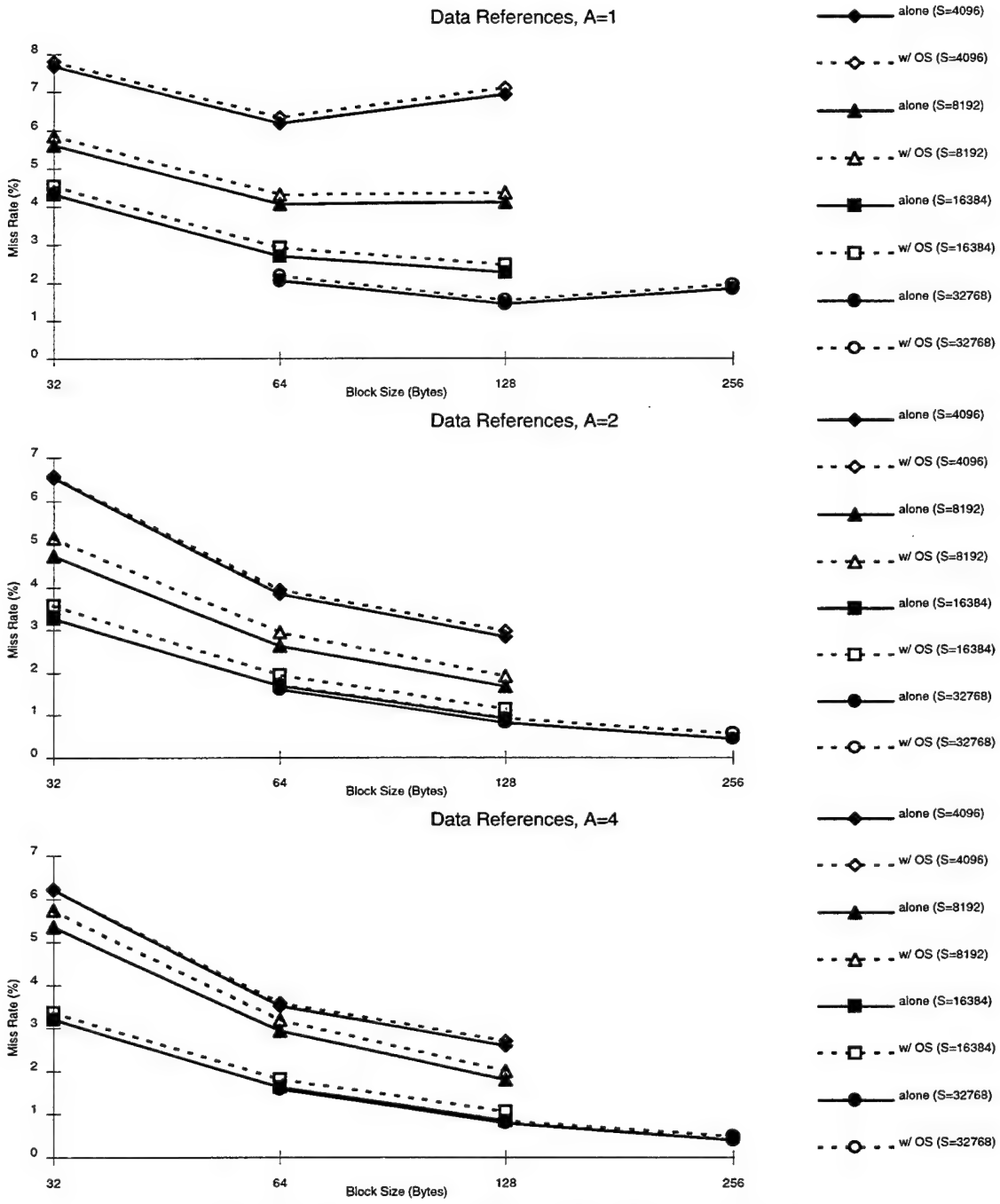

Figure 14: Process Data Reference Miss Rates For Alvinn

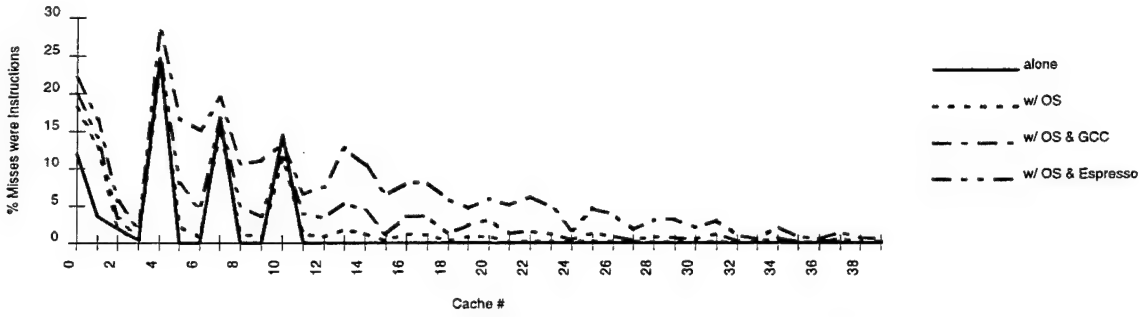

Figure 15: Percent Misses From Instructions, Compress

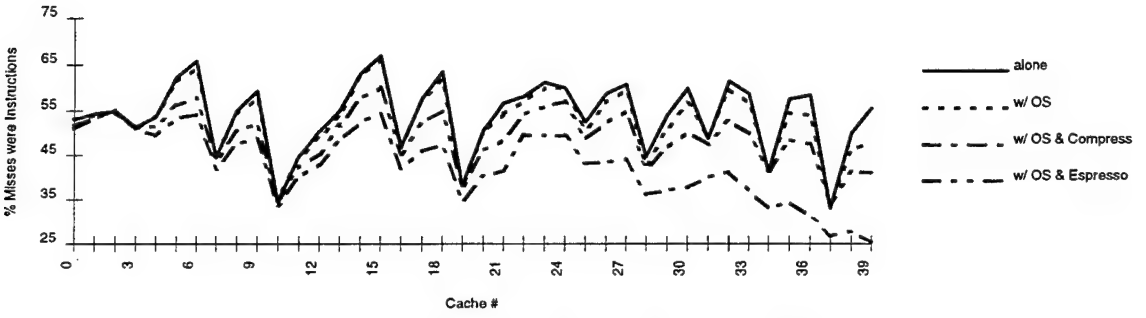

Figure 16: Percent Misses From Instructions, GCC

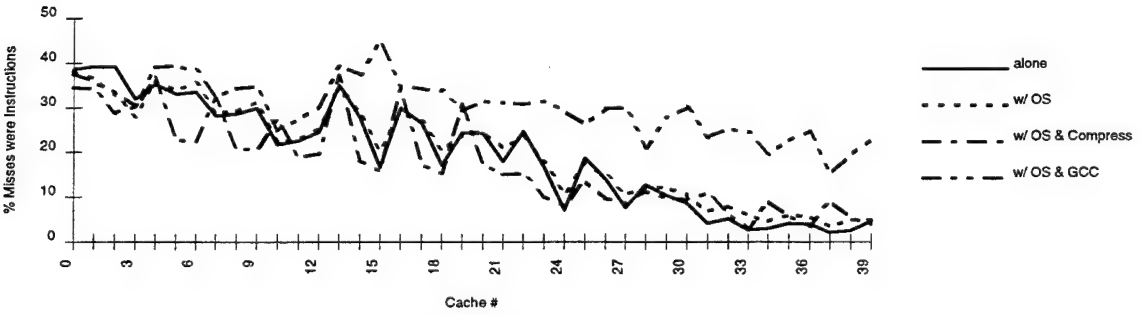

Figure 17: Percent Misses From Instructions, Espresso

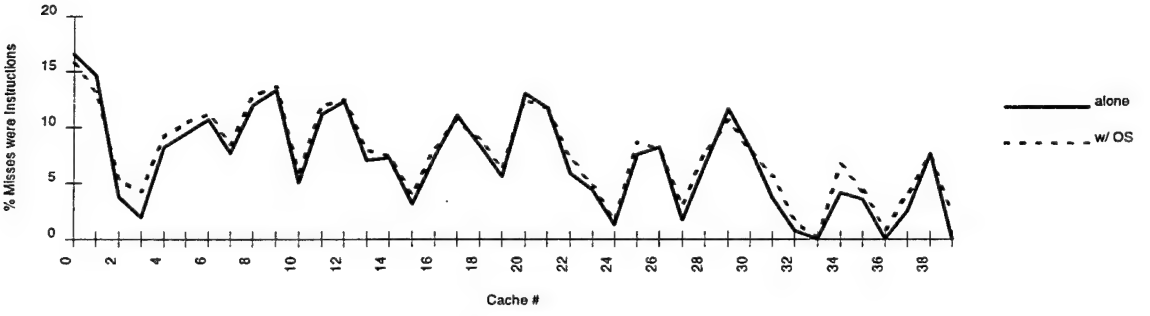

Figure 18: Percent Misses From Instructions, Alvinn

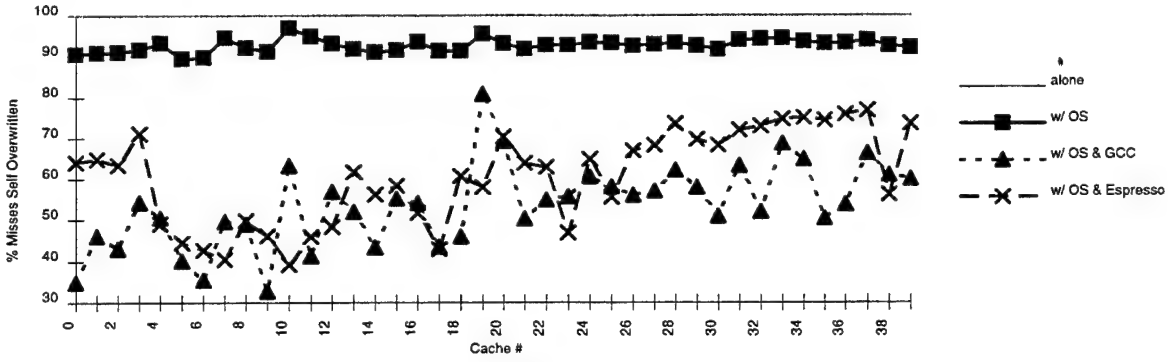

Figure 19: Percent Self Overwritten for Compress

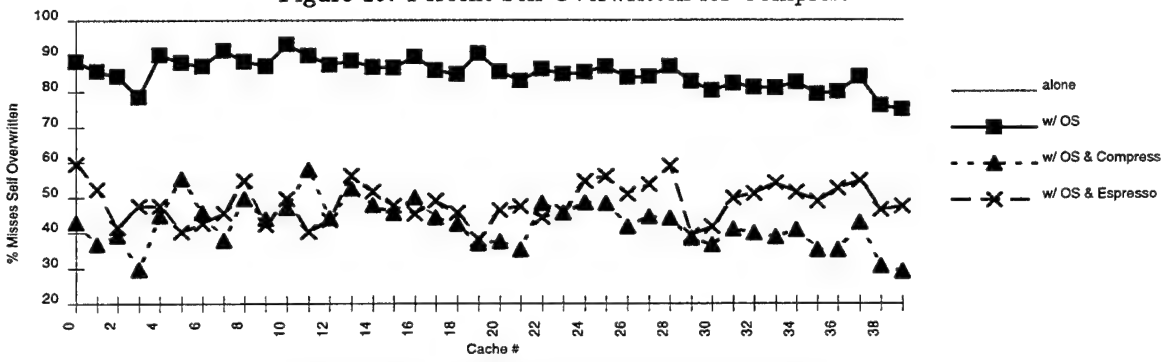

Figure 20: Percent Self Overwritten for GCC

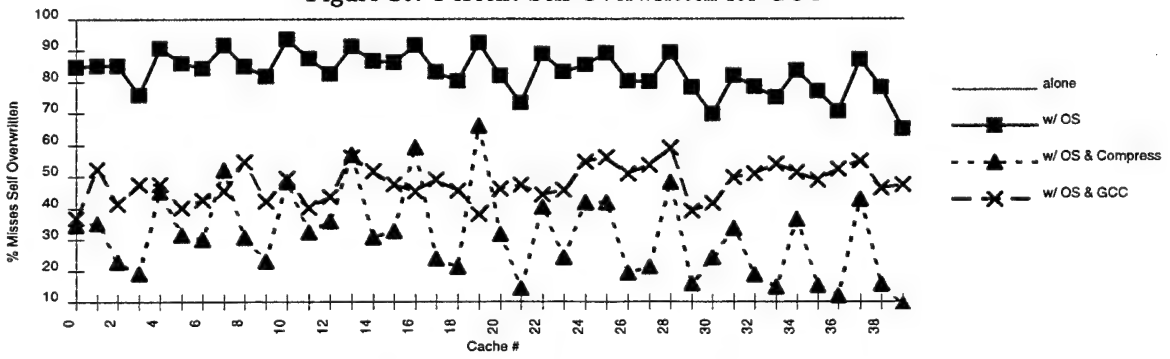

Figure 21: Percent Self Overwritten for Espresso

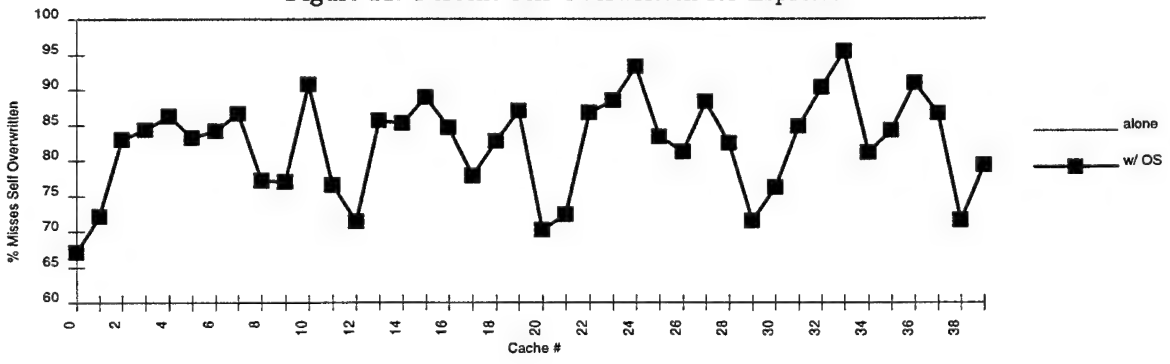

Figure 22: Percent Self Overwritten for Alvinn

5.4 Impact on Cache Performance

So far this analysis has focused on the cache performance within the context of a single program. The impact of the operating system and additional processes is also a factor when the aggregate cache performance is considered, encompassing *all* references from the trace. These results are shown in Figures 23 through 30, which are organized identically to the ones before. The single process simulations for each benchmark are again used as a baseline, with the total cache performance plotted for each simulation that involved that benchmark. Results from simulations with multiple processes are shown in multiple figures, but because all references are considered, the net cache performance is the same regardless of which process is used as the perspective.

The total miss rate is essentially a weighted average of the miss rates of the component processes, as shown below:

$$M = \frac{\sum m_p}{\sum r_p} \tag{1}$$

where M is the total miss rate, m_p is the number of misses for each process, and r_p is the number of references for each process. Because it is a weighted average, the behavior of the total miss rate may be dominated by the miss rate behavior of one of the component processes. A process may dominate the average because of the number of references it generates, such as the combination of a benchmark and its respective operating system overhead (which has fewer references). A process may also dominate the average because of its performance. For example, Compress suffers from particularly poor data cache performance, so any simulation involving Compress will have the average data cache performance dominated by Compress' characteristics. On the contrary, Compress also has the lowest instruction cache miss rates, so the average instruction cache performance is dominated by whatever process is executed with Compress. The dominant process will define the gross performance characteristics of the overall cache behavior. For instance, the miss rate fluctuations as a certain parameter varies, such as cache size.

The impact of each benchmark can be seen by its contribution to the total miss rate, but the impact of the operating system is not as visible. Figures 31 and 32 show the percent of misses that are due to kernel references for instructions and data respectively. As can be seen, the impact to the data cache is much more consistent than that to the instruction cache. The instruction impact varies significantly depending on the benchmark in question and the demands it places on the operating system. Cache design parameters can also be a factor, as the larger caches have a larger portion of

the misses due to the kernel. This is logical as the programs with their larger footprints can take advantage of the larger caches, while the operating system with its shorter execution intervals may never leave the cache warm up phase.

5.5 Summary

Based on the evidence shown here, a few generalizations can be made about the observed cache performance.

- Both operating system and additional user processes will significantly affect cache performance, with the user programs generating the largest impact.
- For a given process, the performance is always degraded due to the external interference, although if the net performance over multiple processes is considered it may be better than the performance for just one of the component processes due to averaging.
- The primary source of this performance degradation is in the loss of temporal locality. The interference between the various processes does not affect each process' spatial locality, but with frequent interruptions in process execution there is a loss of temporal locality across each interruption.
- The worst degradation is in caches which already suffered from poor performance.
- The amount of degradation and any patterns it follows depends greatly on the specific processes involved, and the effects observed can vary greatly. This is due to the differences in program behavior such as system demands (system calls, interrupts) and footprint (size, length, working set).
- The overall cache performance is an average of the performance of the component processes. The individual process performance characteristics are interrelated, so are difficult to determine independently.

This is contrary to some of the initial assumptions made in [1, 2, 3], which have since been discarded. These results are more comparable to those found in [11, 12, 13].

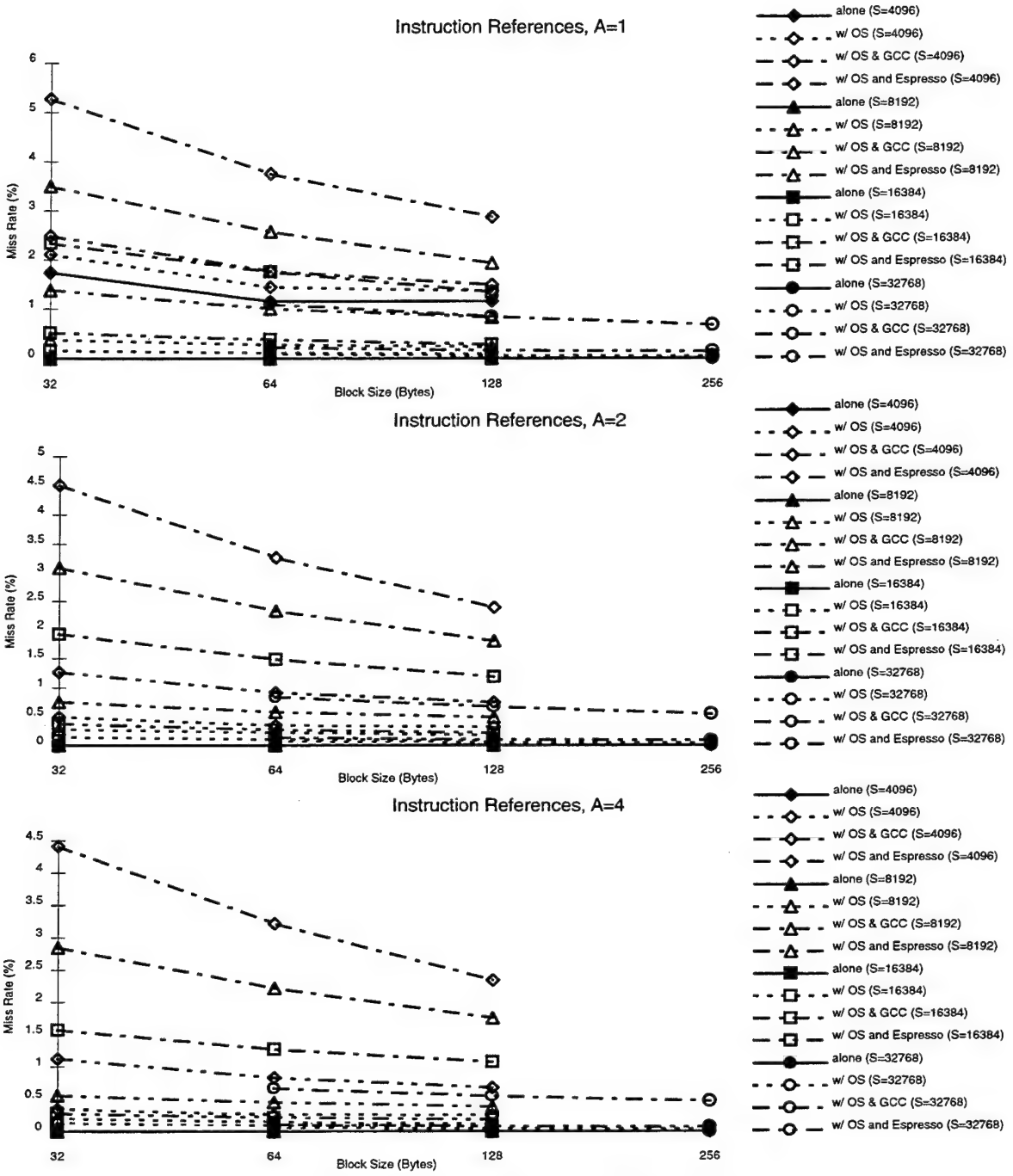

Figure 23: Instruction Cache Miss Rates With Compress

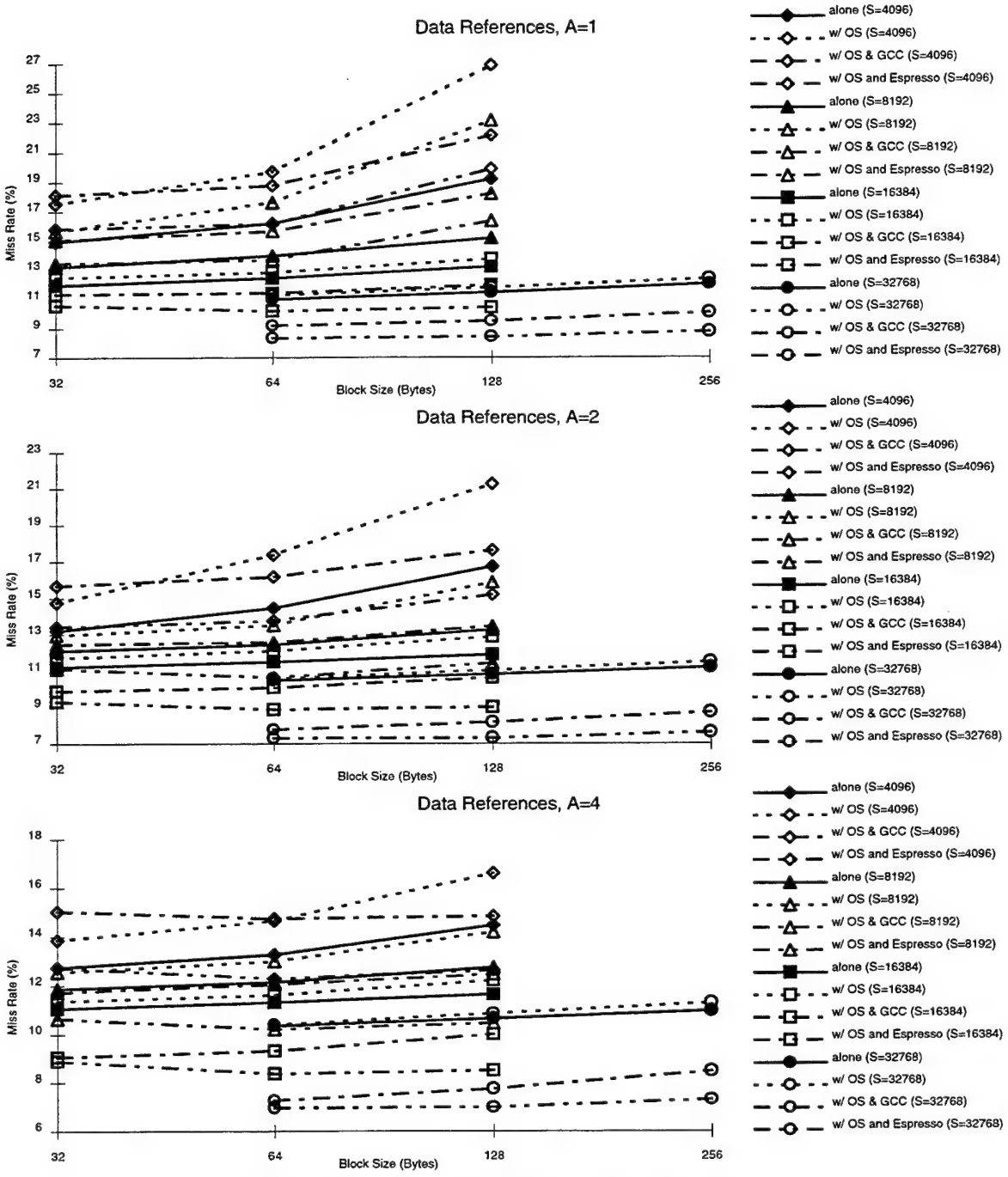

Figure 24: Data Cache Miss Rates With Compress

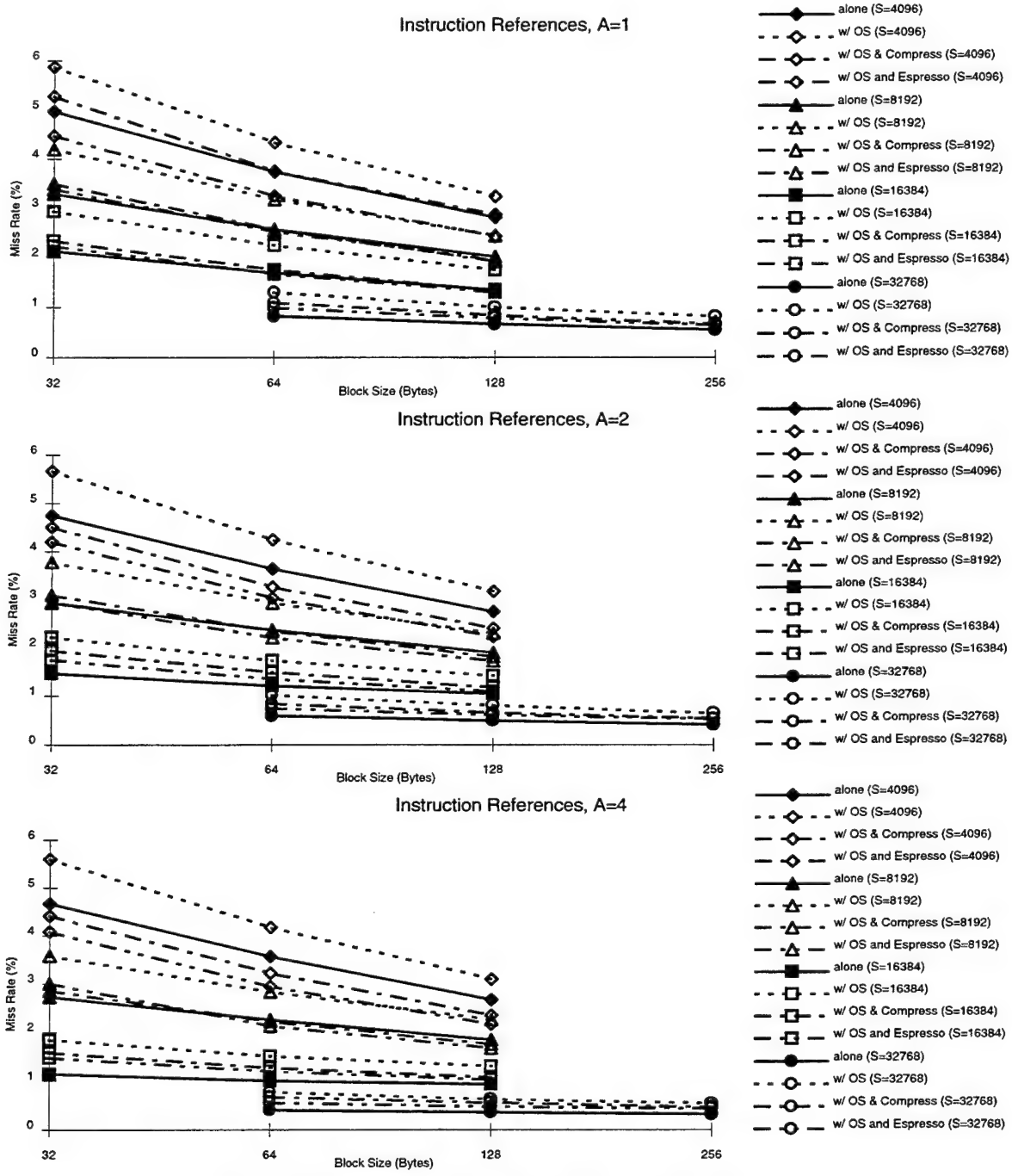

Figure 25: Instruction Cache Miss Rates With GCC

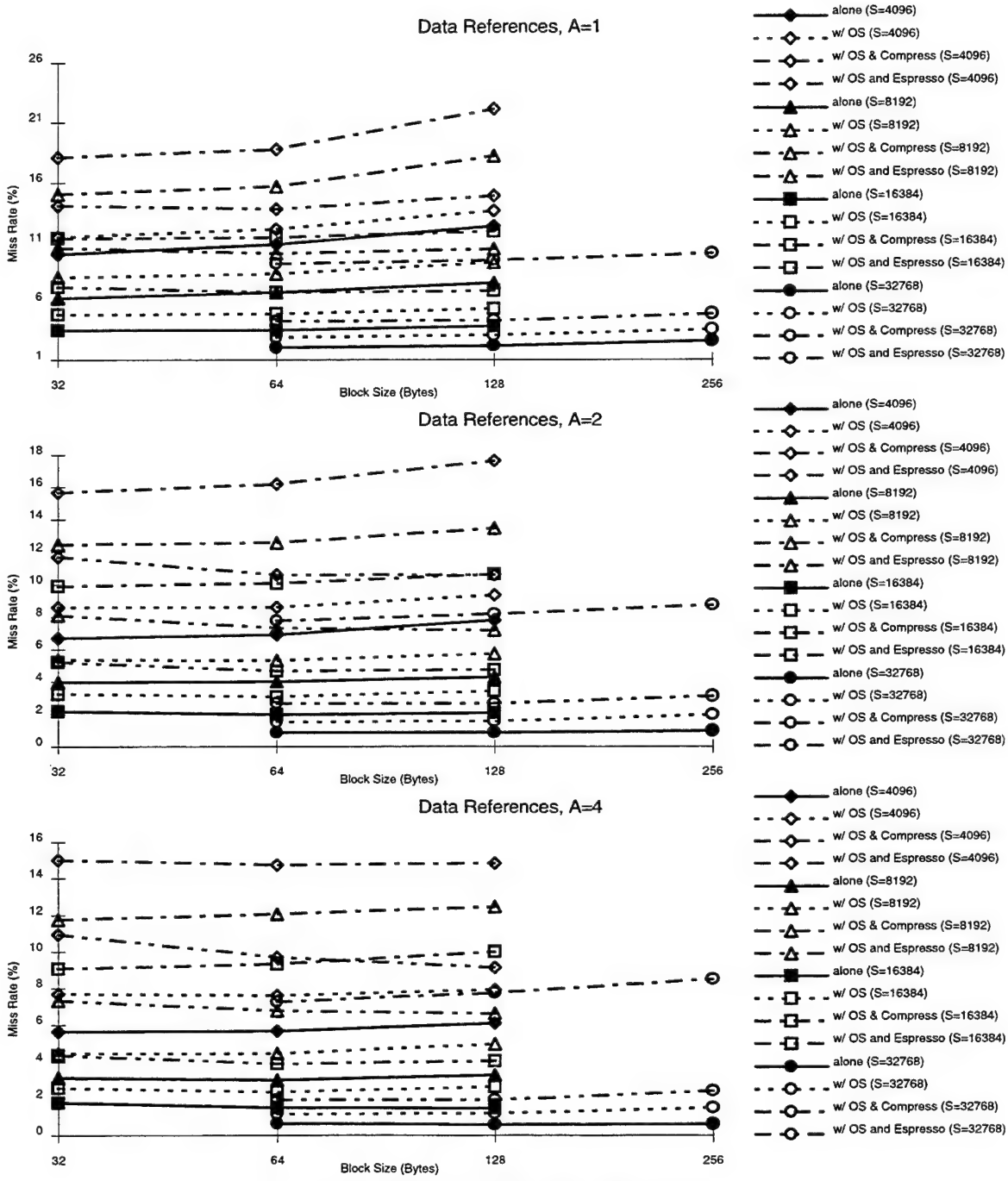

Figure 26: Data Cache Miss Rates With GCC

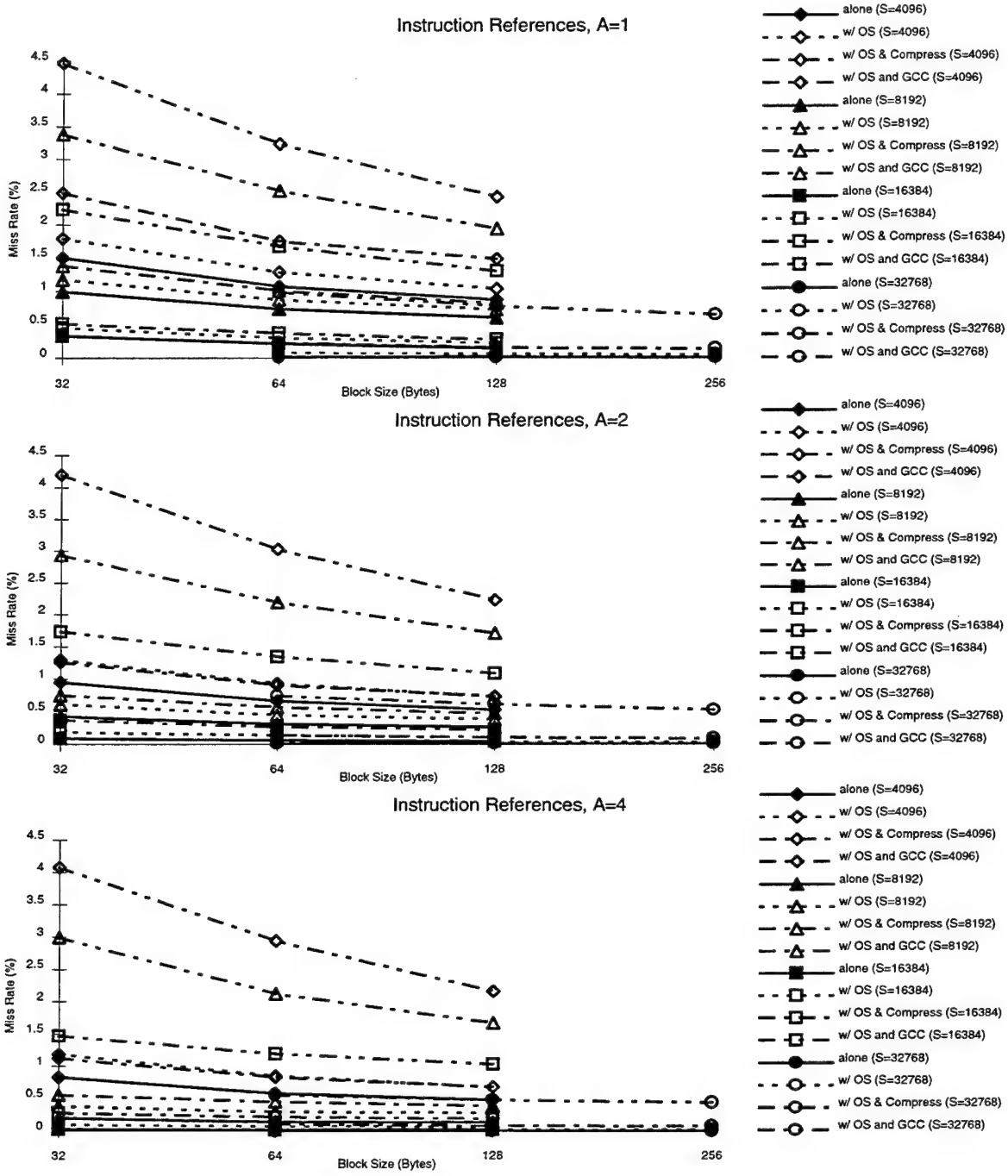

Figure 27: Instruction Cache Miss Rates With Espresso

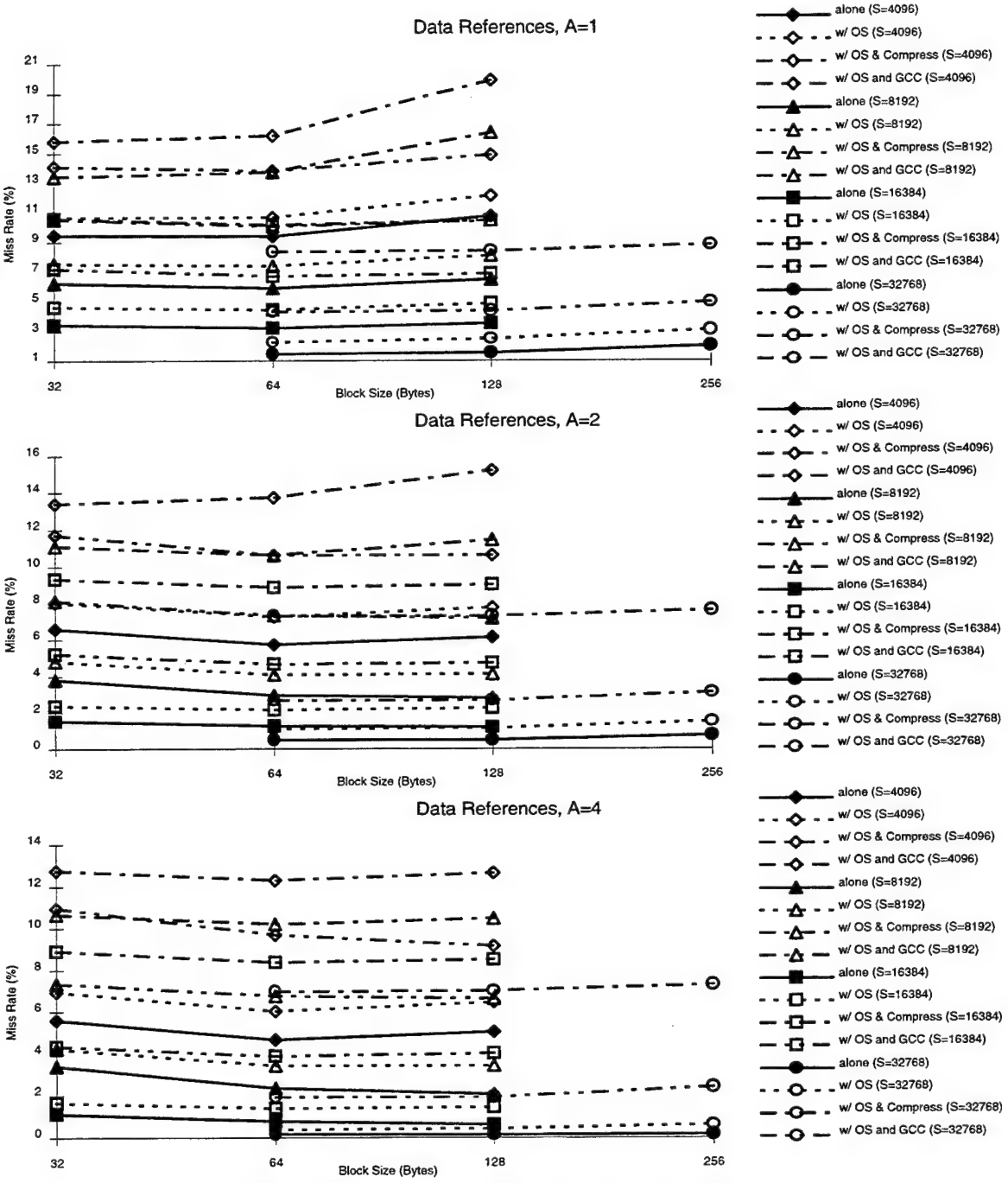

Figure 28: Data Cache Miss Rates With Espresso

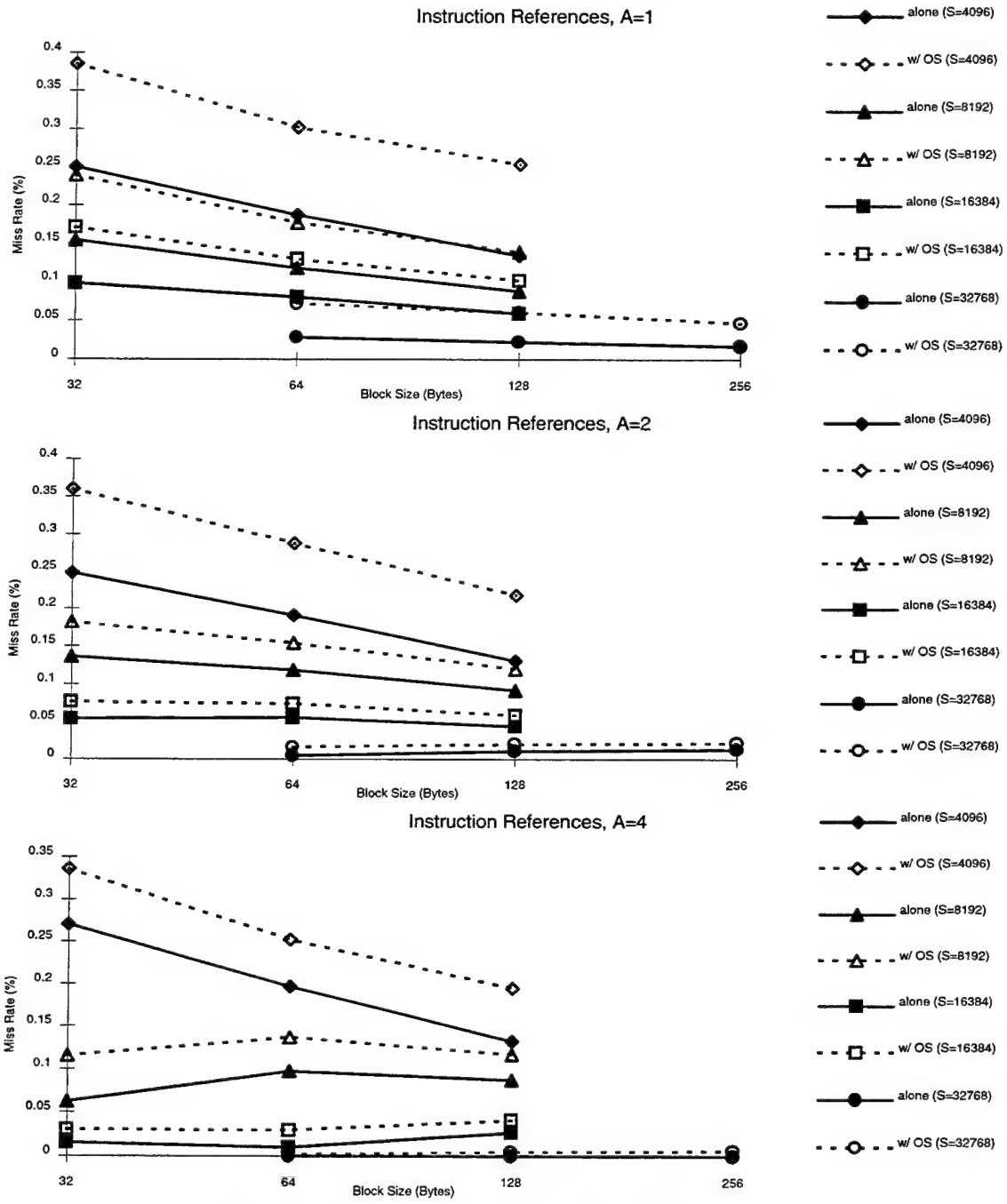

Figure 29: Instruction Cache Miss Rates With Alvinn

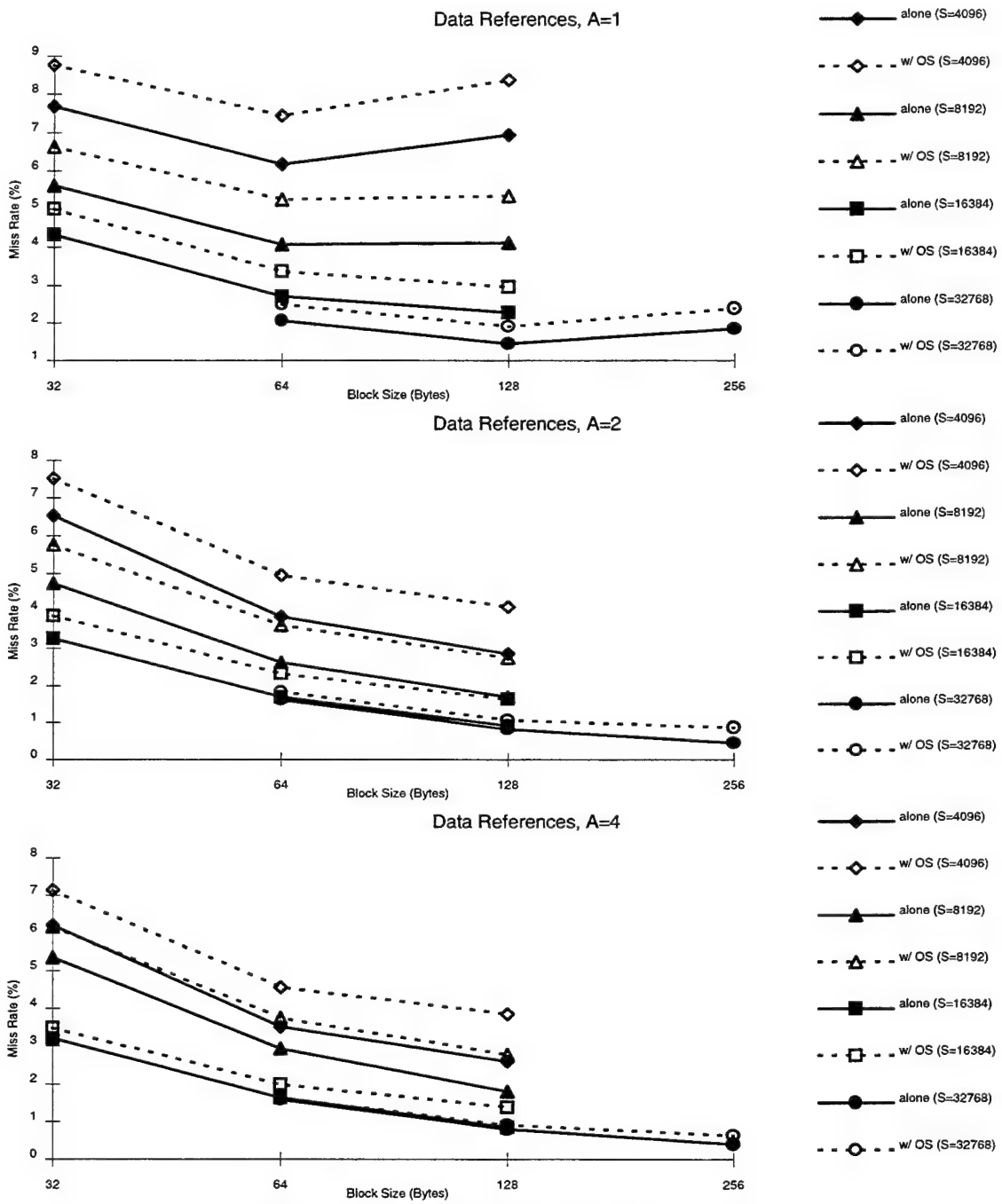

Figure 30: Data Cache Miss Rates With Alvinn

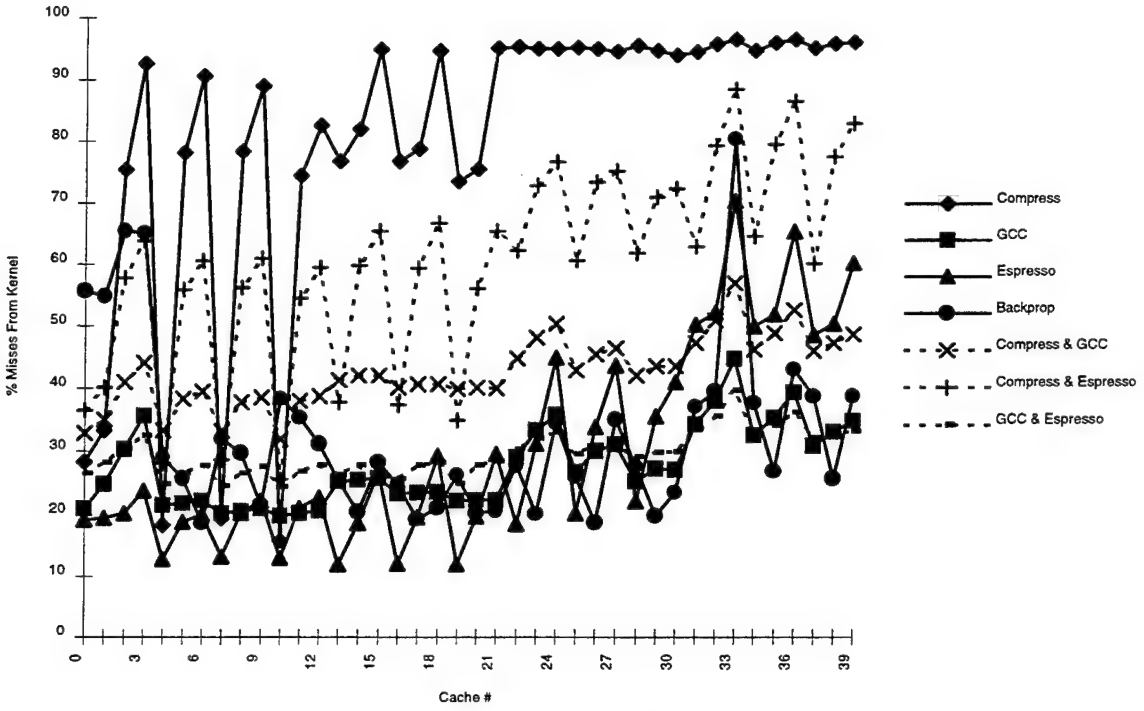

Figure 31: Percent Instruction Misses From Kernel

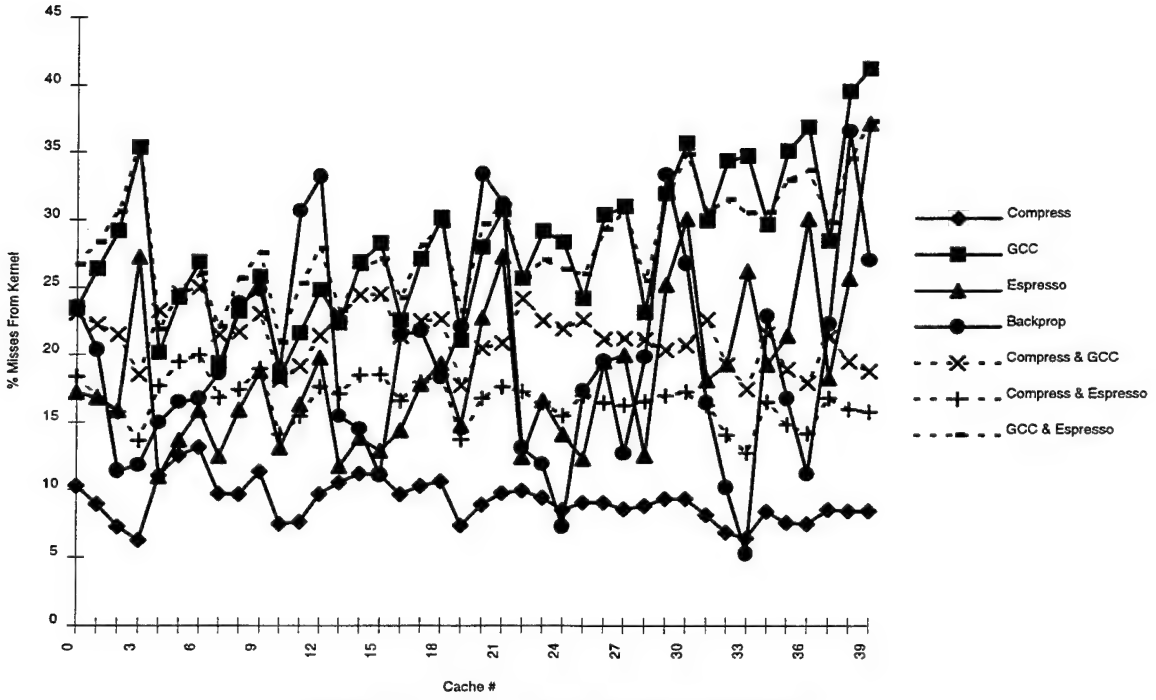

Figure 32: Percent Data Misses From Kernel

5.6 Future Work

With the simulations already performed, there is still a considerable amount of data analysis that could be performed, as more specific aspects of cache performance are considered. Also, a number of improvements to the simulation program were outlined in section 4, which should ideally be included before any future work is performed with this tool. The most fundamental change should be towards modeling more of the memory system, to include such aspects as traffic to memory, physical address mapping, write policies, and cache service times. Other additions can be readily made to the cache simulator to study specific aspects of cache design, such as alternative replacement algorithms in associative caches, different address hashing algorithms, or pre fetching possibilities.

Other more substantial changes could be made to generate different forms of performance data. One area is analyzing sampled cache performance, looking at cache performance over shorter time periods to study the effects of short term working set changes. Another area is tracing the operating system in particular, capturing data from the various kernel threads separately, as well as determining the source of system calls. Another possibility is to provide a more detailed reference record so that reference gap information is available to study interference patterns in more detail. On the most generic level, such a tool can also be used to generate traces for other work. Finally, this research will provide the background necessary for continued study of the operating system through the development of new ATOM tools.

6 Context Switch Model

6.1 Theory

With ATOM, it is now possible to generate simulations with a broader scope than just a single process. As a commercially available tool with a great deal of flexibility, ATOM is simpler to use than past methods, but it still requires a significant amount of additional time and resources to perform the cache analysis. An improvement would be to approximate the accuracy of a comprehensive simulation without the additional effort. One possible method is to develop a synthetic model which would generate complex traces without the execution of programs. Such a technique would exercise the entire cache like a real environment, but is difficult to verify and is beyond the scope of this work.

A simpler method is to study a single, more focused, aspect of cache performance. Here we only consider the performance of a single process, but in the context of a multi-process environment, similar to that considered by Agarwal in [3]. Instead of an entire synthetic workload, an analytical model can be used in conjunction with a single process trace. In this way, the cache behavior of a single process can be predicted more accurately with only a simple simulation. The model is responsible for injecting the desired multi-process characteristics into the simulation, which can be achieved through a statistical approach.

The simulation of a single process will identify its own characteristics, and the introduction of the statistical model will incorporate the transient effects of a complex environment. This can be achieved by analyzing the effect of the operating system and additional processes on a single process, and mimicking this in the simulation program. As will be seen, this is essentially modeling context switch characteristics in the cache [31, 41, 56]. Though it will not be as accurate as the full simulation, it will be faster and much easier to execute. For an approximate result, it is much more efficient.

From the perspective of a single process, it is the sole user of the cache at any given point in time (assuming a uniprocessor environment). However, the time the process is actually being executed is not continuous for its entire lifetime. The process is instead broken up into shorter continuous segments separated by context switches. Between these segments, operating system routines or other processes are being executed, which can overwrite some or all of the process' cache blocks. Assuming all the various processes are independent, these interruptions are transparent to

any single process and each process is not "aware" of the other processes being executed. Here the term interruption is used to denote the time from when a given program is switched out of execution to the point it is returned to execution. The net effect to the cache is that from a specific program's perspective, it is executed continuously, but at certain times during its execution some or all of its cache blocks are overwritten or invalidated. Figure 33 shows the difference between this perspective and the actual environment, showing a basic time space diagram of process execution. This would be the condition in a multitasked uniprocessor where each thread or program is considered to be a unique process with a unique reference stream.

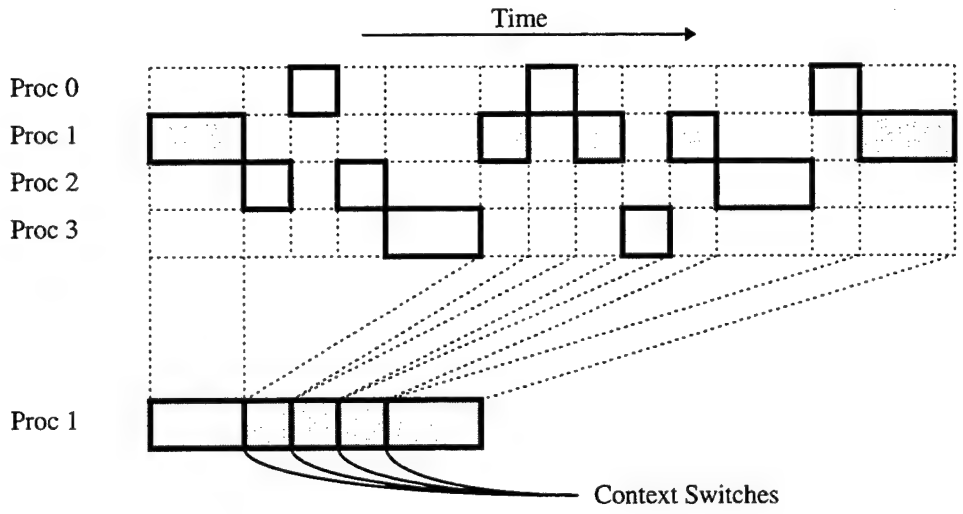

Figure 33: Time Space Diagram of Process Execution

This would suggest that by modeling context switches, the gap between single and multiple process simulations can be bridged. There are basically two fundamental questions that must be addressed by such a statistical model:

1. how often the execution of a program is interrupted by a context switch, and

2. what is the impact to the cache state caused by this interruption.

These questions are not easily answered. Timing of context switches can depend on many variables including the physical system state, how the system is loaded, and characteristics of the programs. Similarly, the impact will depend on the state of program execution, the amount of live data present in the cache, and the amount of overlap, if any, between the working sets of the various programs. The model will depend heavily on the particular system involved, and must be developed with both the

hardware, operating system, and test programs in mind. Once these factors are understood, they can be incorporated into the simulation program so that simulations would theoretically provide results comparable to the program being executed in a realistic environment [23].

6.2 Development

The first step in developing the model is to ensure that it is applicable to our test system [17, 39, 65, 69]. Our Alpha based system meets the criteria described above. It is a single processor machine running OSF, which can execute multiple processes on a timesharing basis. Instructions and data can be shared between processes, but their dependence can be minimized by choosing appropriate test programs. The impact of the test platform on the traces is assumed to be consistent across all simulations and is ignored. The references generated are 64 bit virtual addresses in a continuous address space, so no adaptation of the simulation model is necessary.

Understanding the operating system is the most important aspect of developing the model [4, 9, 18, 70, 72, 71]. The operating system both generates its own set of references, as well as controls the scheduling of the other reference streams. The OSF/1 operating system is a threaded collection of processes which includes system calls, interrupt handlers, and other overhead management/control routines. These can be modeled simply as a collection of additional processes of varying length that are executed at random intervals. The processes are switched in and out of execution just like the test programs. The priority of these processes would require that they occur at any time, preempting the execution of the test process. The various threads that make up the kernel are not independent, and may share substantial amounts of data. By considering the threads of the kernel collectively as the operating system overhead, as was done in the earlier simulations, the model can neglect this shared data with minimal loss of accuracy. The remaining issue is the degree of data sharing between the program and the operating system, which is difficult to pinpoint. For the purpose of this model, this dependence is assumed to be minimal and is neglected, which is a reasonable assumption for the choice of benchmarks. Any simulation of threaded programs or other programs which use substantial cross process communication cannot use these simplifying assumptions.

Given that this type of model is applicable to the simulations already performed, our next task is to analyze the system and program characteristics to define the model's structure. A context switch mechanism must be introduced into the simulation, and the effects of each interruption in execution incorporated appropriately.

6.3 Implementation

One of the most basic forms of modeling multiprocessing is to totally flush the cache at regular intervals, modeling the effect of context switches between processes executing in a round robin fashion [3, 21, 56]. This is realistic for a virtually addressed cache without process identifiers, and a reasonable approximation for a small cache when a context switch will probably overwrite all data, but not appropriate for larger caches when data survival is likely. A more accurate and versatile model is necessary, but will be more complex.

For a model to be effective, however, it cannot be so complex that direct simulation becomes a better alternative. If a detailed description of the test program is required just to develop the model, then simulation may be just as effective. It is also important that the model directly relates to the system it represents. In [31], a very comprehensive model is developed. Unfortunately, it requires a thorough analysis of the program trace to define the model parameters, thus limiting its usefulness. Also, it fails to consider some very basic variations in cache architecture. A balance is necessary, the model must be complex enough to be accurate, but based on basic properties of the system and programs that are easily observed. With this in mind, the model can be developed by answering the two questions mentioned above.

6.3.1 Frequency

The answer to the first question is based on the execution interval of a program, or how long it is executed before a context switch occurs. This is heavily dependent on how execution is scheduled, which is controlled by the operating system [19]. A process is executed until it either is switched voluntarily (i.e., while it waits for some system resource, or requests a system call), it is preempted by a higher priority process (i.e., an interrupt service routine), or it is switched involuntarily for another user process (i.e., the end of a fixed time allocation is encountered). The initial priority of a process depends on its type (system versus user) and its requirements (interactive versus compute intensive). The priority can degrade while the process is being executed and is promoted while it is stalled, which prevents a single process from dominating the system resources. In a fixed priority scheme, processes of equal priority are processed according to a policy, either first in first out (the program executes until completion) or round robin (programs are switched after a fixed interval, taking turns) [16, 71]. The time sharing in OSF/1 is on a thread basis, however the

test programs are all single threaded, and the various threads of the operating system are considered as a conglomerate from the cache's perspective.

For the model, we use a basic scheme based on this information. We assume that all operating system level processes have a higher priority than any test program process, so they can interrupt test program execution at any time. These processes will include both interrupt service routines and system calls. All test programs run at the same priority, with a round robin scheduling. For a single program, this defines the characteristics of its execution interval. The interval has some maximum value where a context switch is automatic, but up to that point there is some probability that a switch will occur earlier due to either an interrupt, system call, or stall waiting for resources. Based on results from previous studies [8, 31, 41], this probability follows an exponential distribution. Most processes execute for a short interval; with an exponential reduction so very few processes consume the maximum interval — showing that context switches are a regular occurrence. With round robin scheduling, the number of test programs considered in the model does not affect the execution interval.

To incorporate this fact into the model, a random variable R is defined representing the execution interval length in number of references r with an exponential probability density function. A distribution of this kind has the form [53]:

$$f(r) = \frac{1}{\mu} e^{\frac{-r}{\mu}} \tag{2}$$

where μ is a constant which defines the shape of the curve and its expected value. The probability that any given reference interval R will be r references or less is defined by:

$$P[R \leq r] = \int_{-\infty}^{r} f(r)dr = 1 - e^{\frac{-r}{\mu}} \tag{3}$$

If we assume that an interval will be as long as possible, then this can be used as the probability that a given execution interval R is r references long, expressed as:

$$p = 1 - e^{\frac{-r}{\mu}} \tag{4}$$

This function could be incorporated into the program by determining the probability of a given interval as that reference is reached. A random number in [0..1] is then generated at each reference to determine if a switch is necessary. A better solution is to invert the equation to yield:

$$r = -\mu \ln(1 - p) \tag{5}$$

Thus generating a random number in [0..1] will generate an appropriate execution interval length r (rounded to an integer value), as shown in Figure 34.

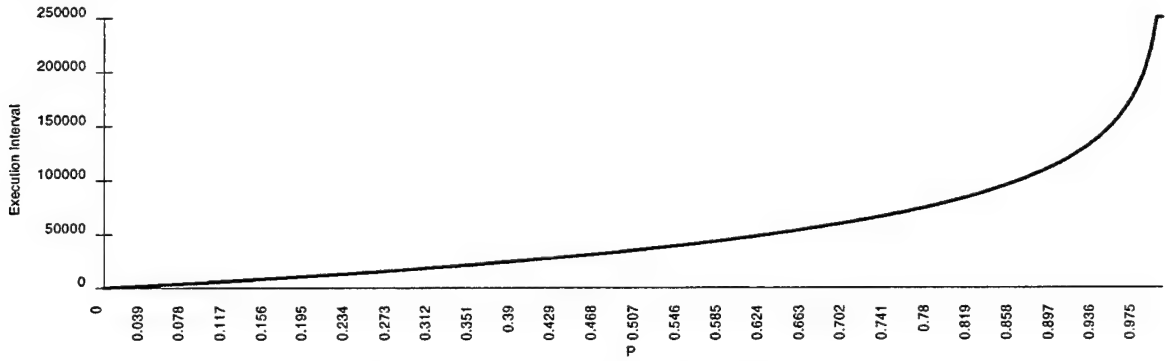

Figure 34: Execution Interval Given Some Probability [0..1]

The remaining unknown is μ, which can be determined by defining the desired maximum execution interval. In [8, 41] this was 400,000 traced instructions, or 25,000 untraced, although these values based on a system that is no longer contemporary. If we assume that each program executes for a maximum 10 ms time slice on a system with a 20 ns cycle and average of 2 cycles used per instruction [71], this generates a maximum interval of 250,000 references:

$$\frac{(10e-3\ \frac{seconds}{interval})}{(2\ \frac{cycles}{instruction})(20e-9\ \frac{seconds}{cycle})} = 250,000\ \frac{instructions}{interval} \tag{6}$$

At this point, the probability of a context switch defined above should approach 1, or

$$\lim_{r \to r_{max}} e^{\frac{-r}{\mu}} = 0 \tag{7}$$

Obviously this cannot be exact, but selecting a μ of $\frac{r_{max}}{5}$ or 50000, is accurate to 0.006738 which is sufficient for this application. Since the exponential function cannot define the maximum value, an explicit limit is set on the function, so that the final definition of each execution interval is given by:

$$r = \min(-50000 \ln(1-p), 250000) \tag{8}$$

which is the function used to generate Figure 34.

Incorporating this into software, at program start and after every context switch, a random value is generated in [0..1]. This is applied to the above function to determine the execution interval. A counter is maintained of the number of instruction references since the last context switch, and when these two values are equal, the switch impact model discussed below is performed. The actual distribution generated by the random function is shown in Figure 35, showing the probability of a

specific interval determined by the number of intervals out of 250,000,000 generated. The probability of any particular interval is low, but the cumulative probability of a context switch as the interval increases to its maximum value approaches 1 as expected. The spike at 250000 references is due to the limit in the function, and is negligible in the cumulative distribution.

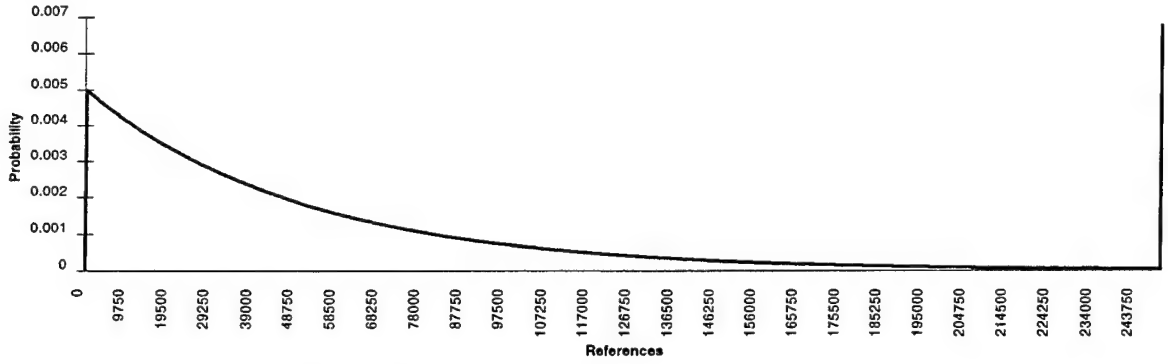

Figure 35: Actual Distribution of Random Execution Intervals

6.3.2 Impact

The second question addresses the likelihood that data in the cache is overwritten by the processes executed during the interruption. As stated before, simply invalidating the entire cache is not a realistic model. Instead, the model must take into account the footprints of all processes executed during the interruption to determine what portion of the cache is overwritten. This is addressed by both Agarwal [3] and Thiebaut and Stone [56]. Both models attempt to evaluate all aspects of the cache analytically. By using simulations, much of the model can be discarded. Instead, only the relevant function regarding the probability of cache line replacement is used. Both papers use identical functions to determine the probability that a program's working set will have a certain number of unique references to a given cache line. The derivation of this function is quite lengthy, for more information please consult either paper. It is based on the binomial probability that any given cache reference will be assigned to a certain cache line.

The calculation is a function of the number of cache lines N, the cache associativity A, and the footprint F of the interruption, defined as the number of unique blocks referenced by the program in the interval under consideration. The probability that a given cache line will contain i references from a certain footprint is defined as:

$$\text{if } 0 \leq i < A \ :$$

$$p_i = (\frac{F!}{i!(F-i)!})(\frac{1}{N})^i(1-\frac{1}{N})^{F-i} \tag{9}$$

$$\text{if } i = A \ :$$

$$p_A = \sum_{j=A}^{F}(\frac{F!}{j!(F-j)!})(\frac{1}{N})^j(1-\frac{1}{N})^{F-j} \tag{10}$$

The second term is not readily computable, so for simplicity it can also be calculated as:

$$\text{if } i = A \ :$$

$$p_A = 1 - \sum_{j=0}^{A-1}(\frac{F!}{j!(F-j)!})(\frac{1}{N})^j(1-\frac{1}{N})^{F-j} \tag{11}$$

The probability that a certain number of blocks will be used on any given line directly determines the probable number of blocks that must be evicted from that line during the interruption.

Unfortunately, this function cannot be inverted to give a direct calculation of the number of blocks overwritten in each line based on a single variable in [0..1]. Instead, a random probability p is generated for each line in each cache and the following algorithm is used to iterate over all values of a in the range $[0..A-1]$ to determine the number of overwrites to be performed on that line:

$$\text{if } p > \sum_{i=0}^{a} p_i \text{ , then } a+1 \text{ overwrites are performed} \tag{12}$$

Based on [56], the overwrites caused by this function follow a roughly normal distribution. Figures 36 and 37 show the probability of n overwrites per line, $P(n)$, for a context switch with interruption footprints of 100 and 1000 respectively. Various associativities and their possible replacements are shown, with the replacement probability plotted against the number of lines in the cache — showing the decreasing likelihood of replacement as cache size increases or footprint size decreases.

Certain assumptions apply to the formulas provided in the papers. These equations assume that a program's footprint is uniformly distributed over the cache. The locality in reference streams would suggest that this is not true, which was supported by the results in both papers. Using other mapping algorithms (hashing), it may be possible to get a more uniform distribution, but this technique was not used. Finally, shared references between programs are neglected. As discussed before, given the test programs used and the way the kernel is considered, this is a reasonable assumptions. To analyze a threaded program, or one with a substantial shared component (such as a database), such an assumption is not valid.

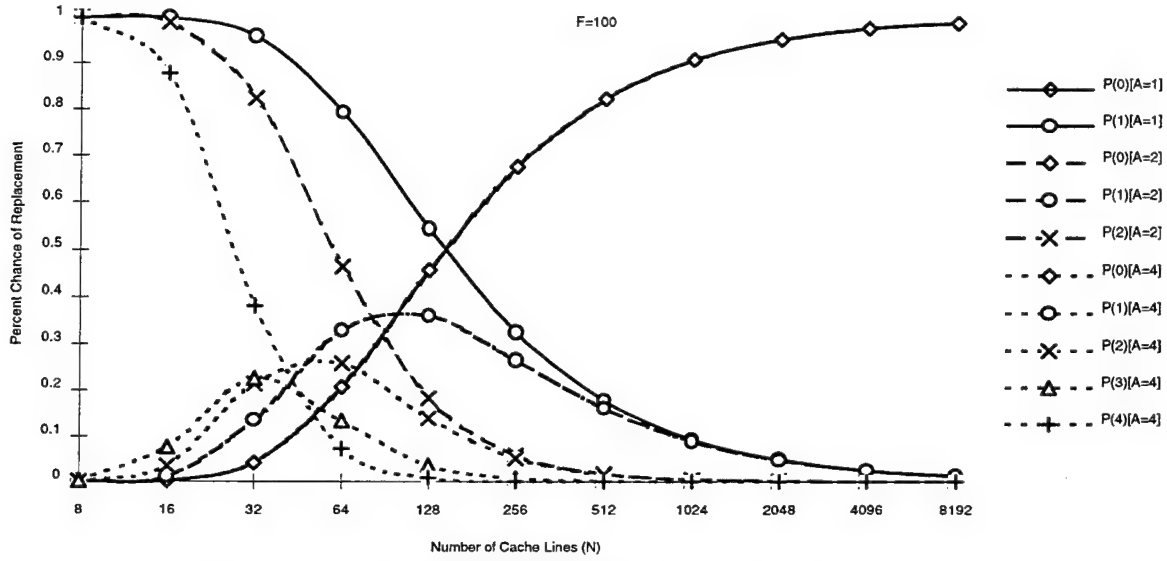

Figure 36: Probability of Cache Blocks Being Overwritten; F=100

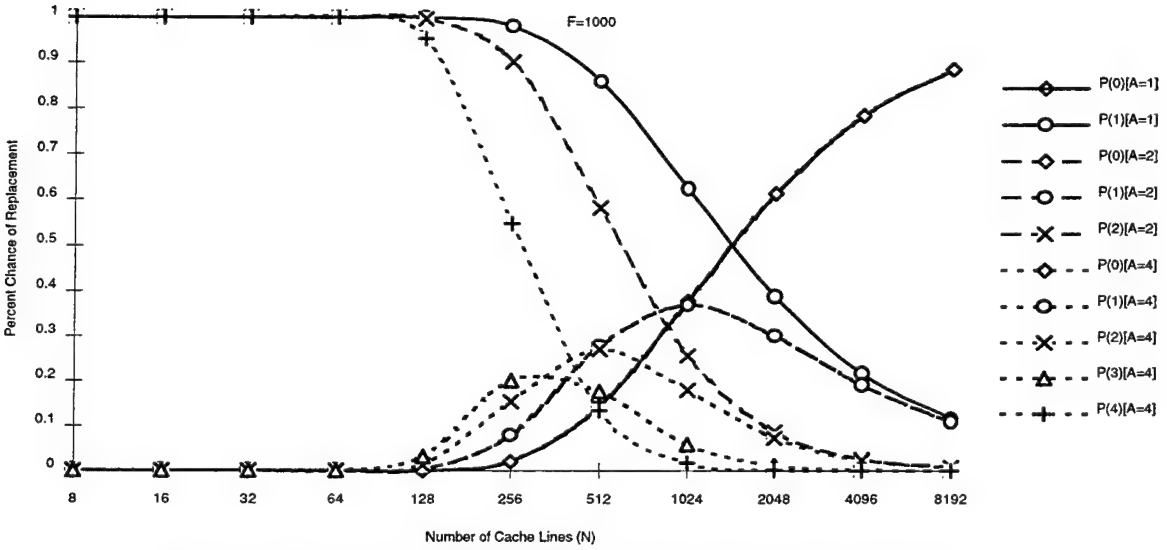

Figure 37: Probability of Cache Blocks Being Overwritten; F=1000

Other assumptions made in the papers are no longer relevant. The use of LRU replacement is assumed in the analytical model, but incorporated explicitly in simulation. The LRU blocks are selected for overwrite, but other selection methods are possible. Also, other considerations such as which cache lines present at a context switch will be referenced after the interruption period do not have to be modeled, since they are determined by the simulation.

The remaining problem is determining the footprint of the interruption. The footprint depends on the process being considered, its state of execution, and the line size of the cache, so is very difficult to characterize. In [3, 56] detailed analyses of program traces were used to determine this value. This is not compatible with our goal of minimal analysis in developing the model, so a different, more improvised, approach is used. Based on the footprint values used in other work [3, 56], a reasonable (though less accurate) range can be achieved using:

$$F_i = \frac{r_{int}}{50 * B} \tag{13}$$

$$F_d = \frac{r_{int}}{50} \tag{14}$$

which gives the instruction footprint as 2% of the execution interval of the interruption (r_{int}) divided by the block size (B) in words (or in bytes divided by 4), and the data footprint is simply 2% of the execution interval. This is obviously an overly simplified approach to characterizing the footprint, but adequate for an initial review. For a unified cache, the two footprints are simply summed, which is correct assuming independence of instruction and data references (no self modifying code). For a range of intervals [0..250,000], this produces a footprint range of [0..5625] for the caches simulated.

The execution interval of the interruption is computed as

$$r_{int} = n * -\mu \ln(1 - p) \tag{15}$$

where n is the number of additional processes being executed according to the model and p is a random value in [0..1] as used before. This is consistent with the round robin scheduling, as the number of processes being executed determines the length of interruption. One problem is that the models used in both [3, 56] neglect the operating system. For simplicity, the operating system is modeled as just another process: to simulate a process with the operating system, $n = 1$; with the operating system and one other process, $n = 2$; and so on. This may be pessimistic, as one might expect that system calls and interrupt service routines to be shorter than user programs, however the distribution of execution intervals is weighted towards shorter intervals, which is consistent with frequent interruptions.

The impact is applied in software every time a context switch is indicated. The length of the interruption is computed, which in turn defines the footprint for the various unified, instruction, and data caches. This is used to calculate the probability that a given number of cache blocks are overwritten for each cache line in each different cache configuration. Then for each cache line a random number in [0..1] is generated and compared to the probability to determine how many blocks on that line (up to the set size) are invalidated.

6.4 Testing

The mechanism described above was incorporated into the same program used for the single processes simulations described in section 5. The additional code is also included in appendix A. Again a tool was defined to instrument the test programs (called `mod`) so shared library functions could be used in analysis. Simulations with the model were performed using the same 40 caches on all four benchmarks for $n = 1$, modeling the program with the operating system. Simulations were also performed for $n = 2$ for Compress, GCC, and Espresso, to compare the model results to simulations of two concurrent processes with the operating system. All simulations were performed on the same Alpha system as before. The results of the model simulations are reviewed in the next section, and compared with their equivalent "real" simulations.

7 Model Evaluation

7.1 Individual Results for n=1

The accuracy of the context switch model can be seen in its ability to predict cache miss rates commensurate with those generated from an equivalent "real" simulation. The first test case was for n=1, modeling the test program with one additional process, the operating system, which was performed for Compress, GCC, Espresso, and Alvinn. The results of these simulations are plotted against the corresponding real simulation of each program with the operating system, shown in Figures 38 to 41.

As can be seen, the model generally provides an adequate mechanism for predicting the interference caused by operating system overhead. There are some variations over the results, although certain instances such as Alvinn data references are quite accurate. Such variations are to be expected given the assumptions that were used to generate the model. The only significant fluctuations occur for Compress, which is logical considering that benchmark interacts substantially more with the operating system than the others.

7.2 Individual Results for n=2

A better test of the model is for n=2, modeling the effects of the operating system and an additional process on the performance of the test program. Simulations were performed for Compress, GCC, and Espresso; Alvinn was neglected since no corresponding real simulation could be performed. These results are shown in Figures 42 to 44.

These results show the weakness of the model. In almost every case, the model predictions are more optimistic than the real data. Also, the model does not account for differences in program behavior, so while there are two sets of real data from two alternative second programs, the model only predicts a single result. Based on this, the model does not accurately predict the amount of interference generated from multitasking. The error in the model should also be more pronounced as the level of multitasking is increased, but no simulations could be performed with 3 test programs or more to verify this.

7.3 Interference Comparison

The primary source of error in the model is apparent in the interference plots. These are equivalent to the interference figures of the previous results, showing what percentage of cache misses

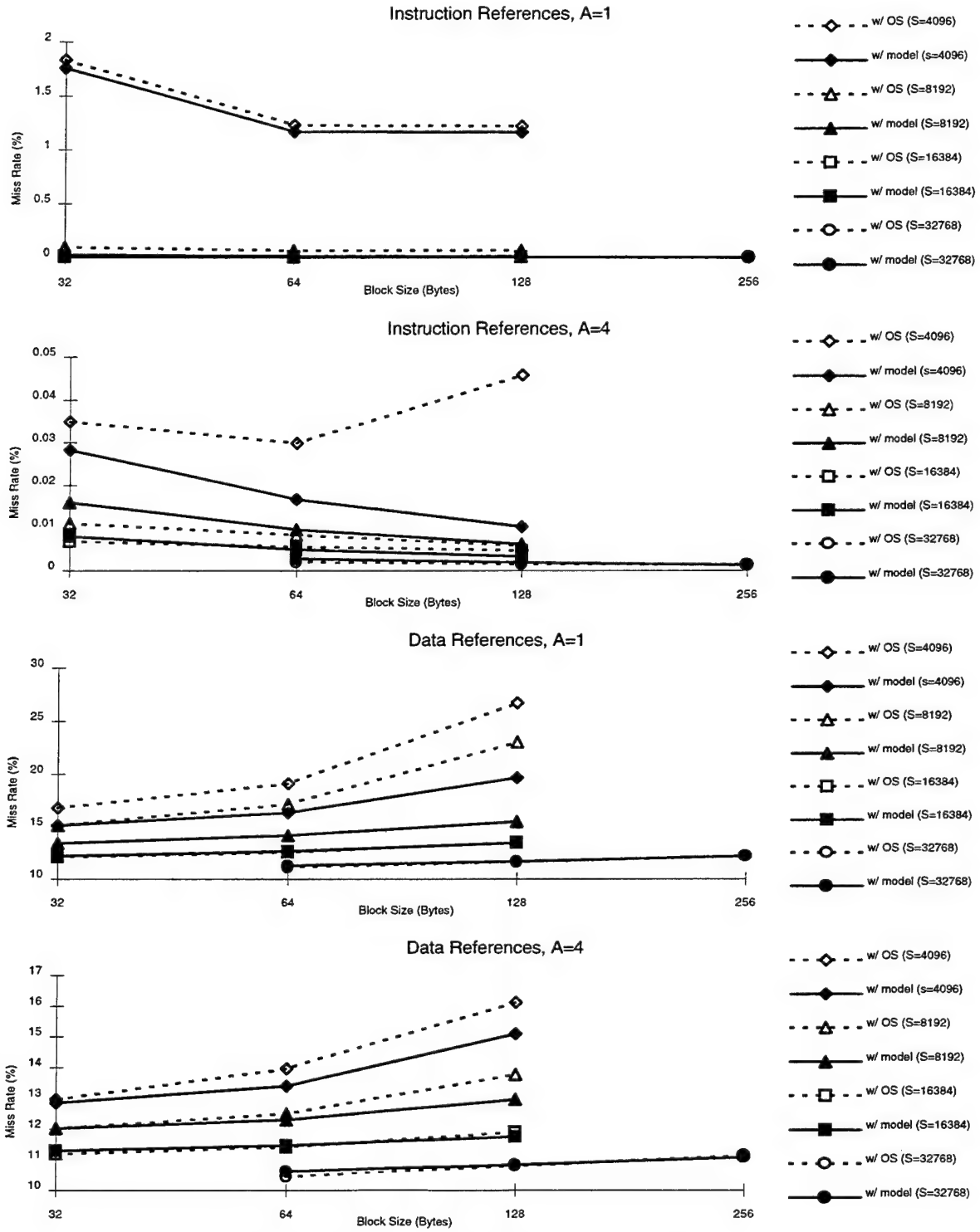

Figure 38: Model Results for Compress; n=1

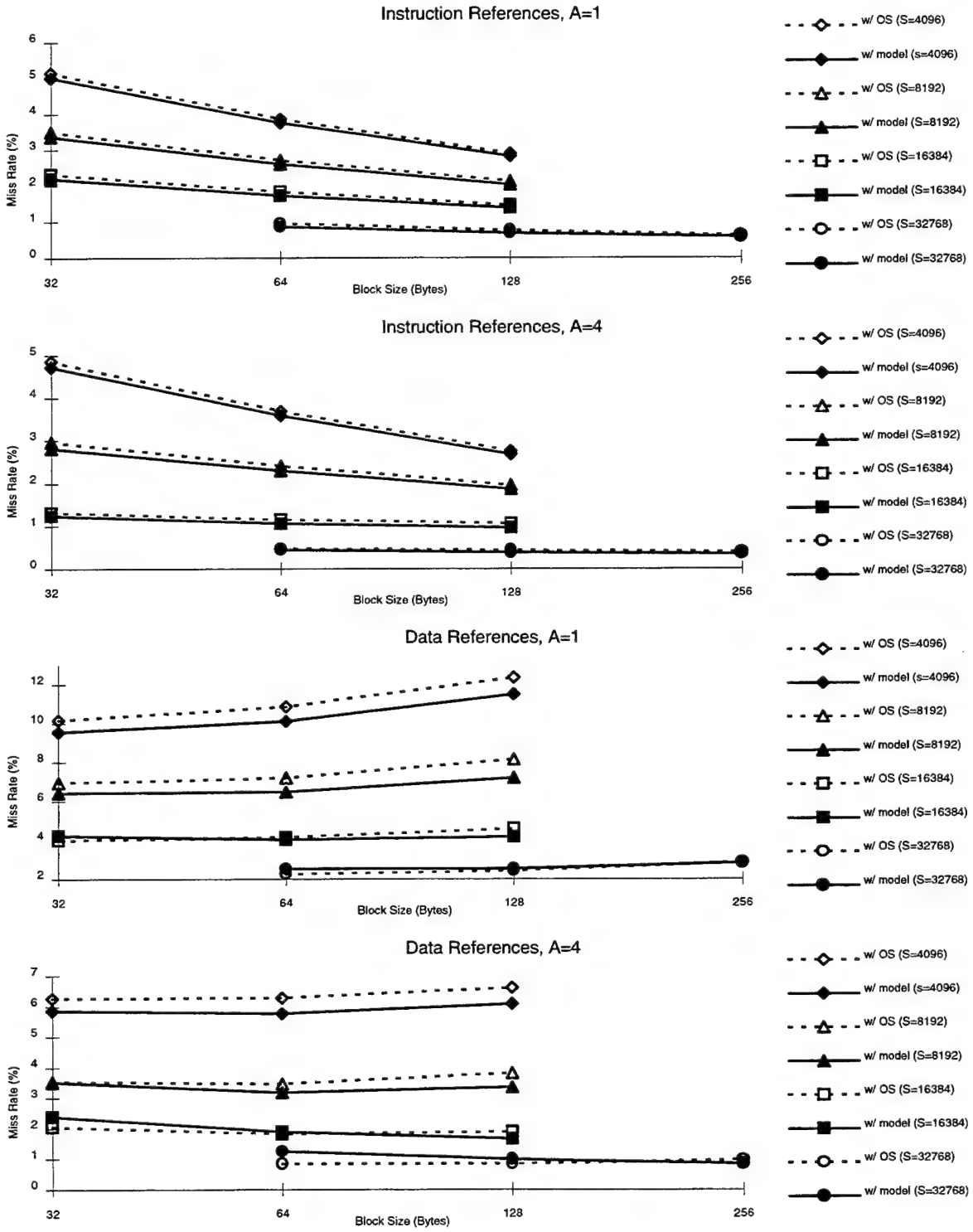

Figure 39: Model Results for GCC; n=1

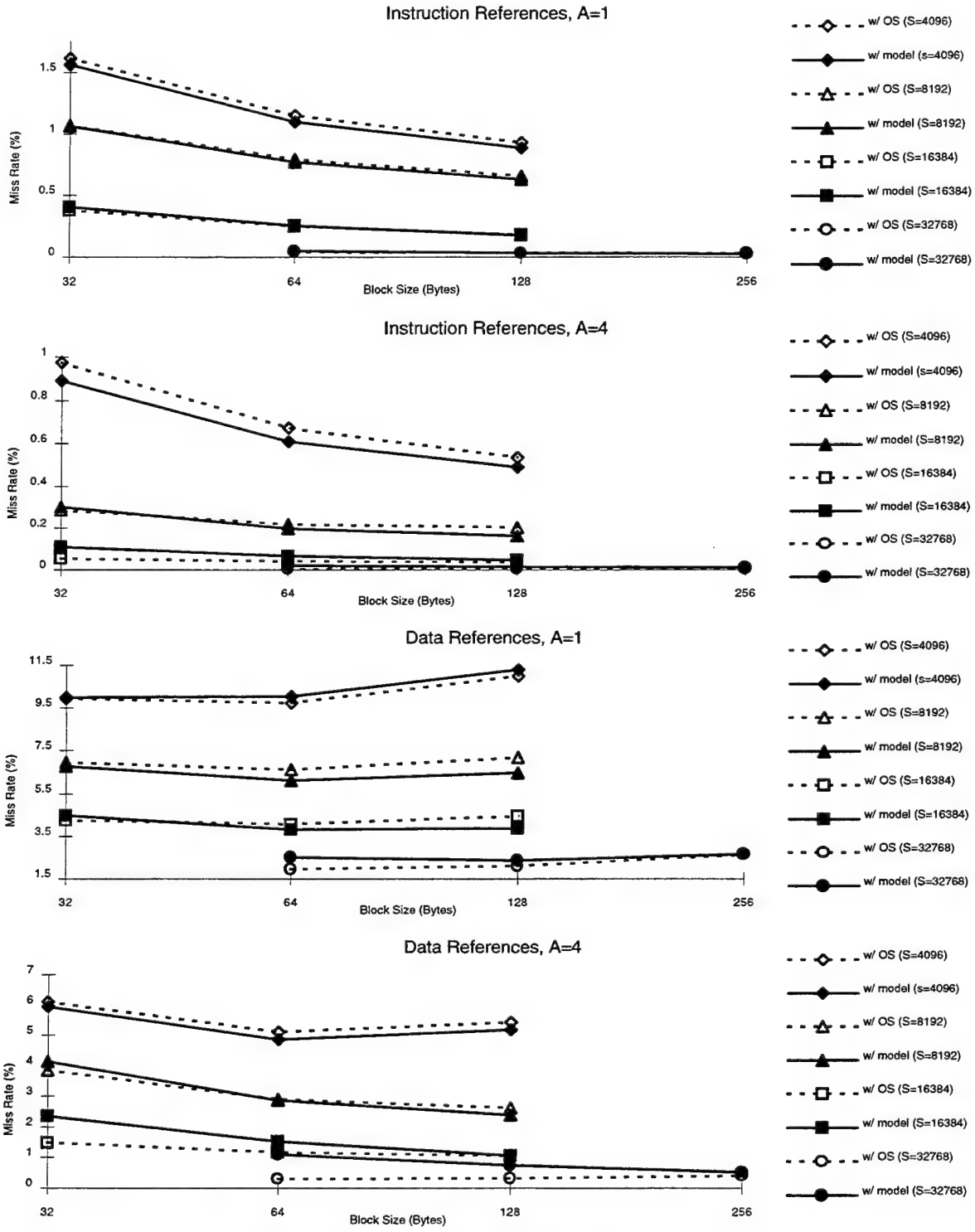

Figure 40: Model Results for Espresso; n=1

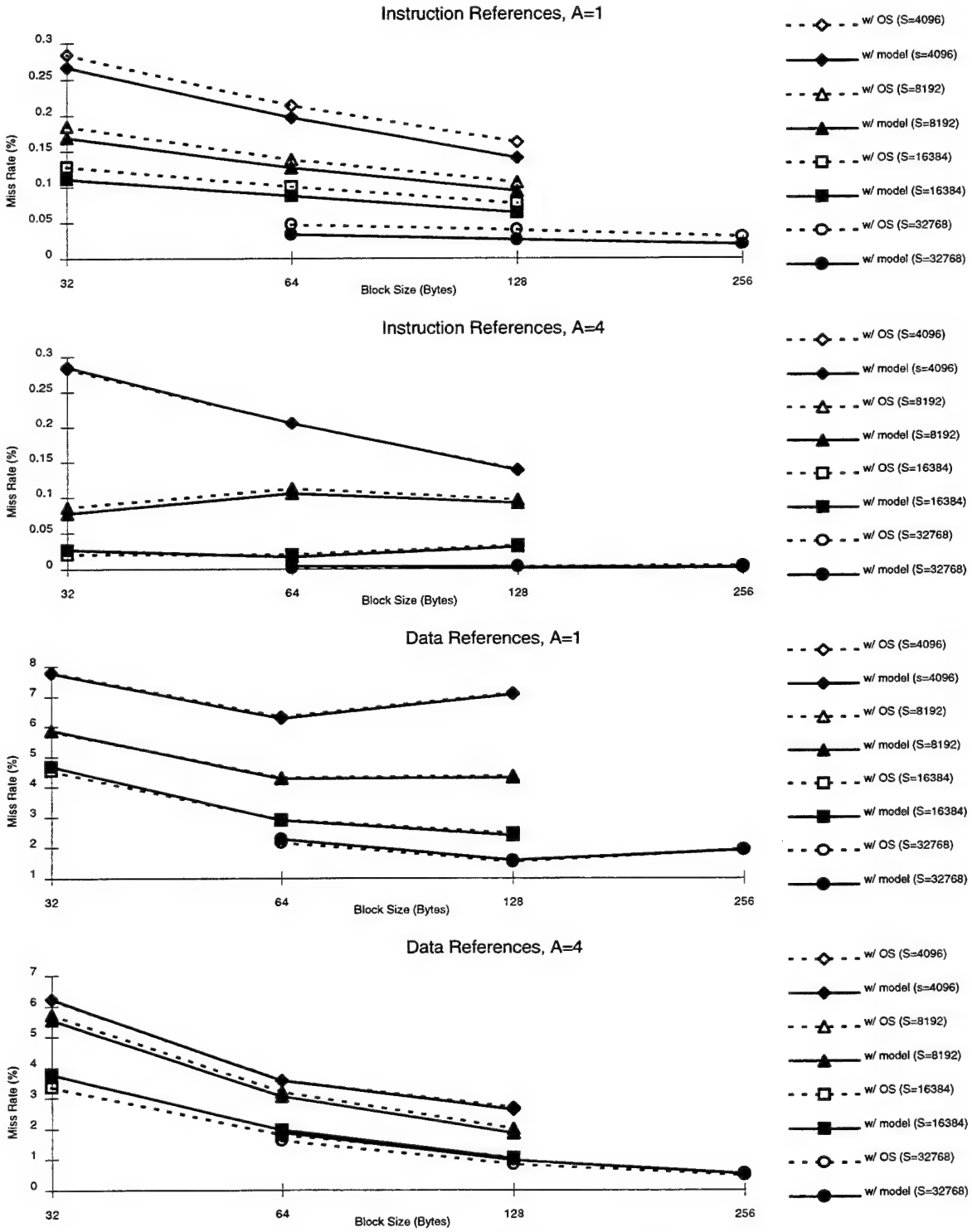

Figure 41: Model Results for Alvinn; n=1

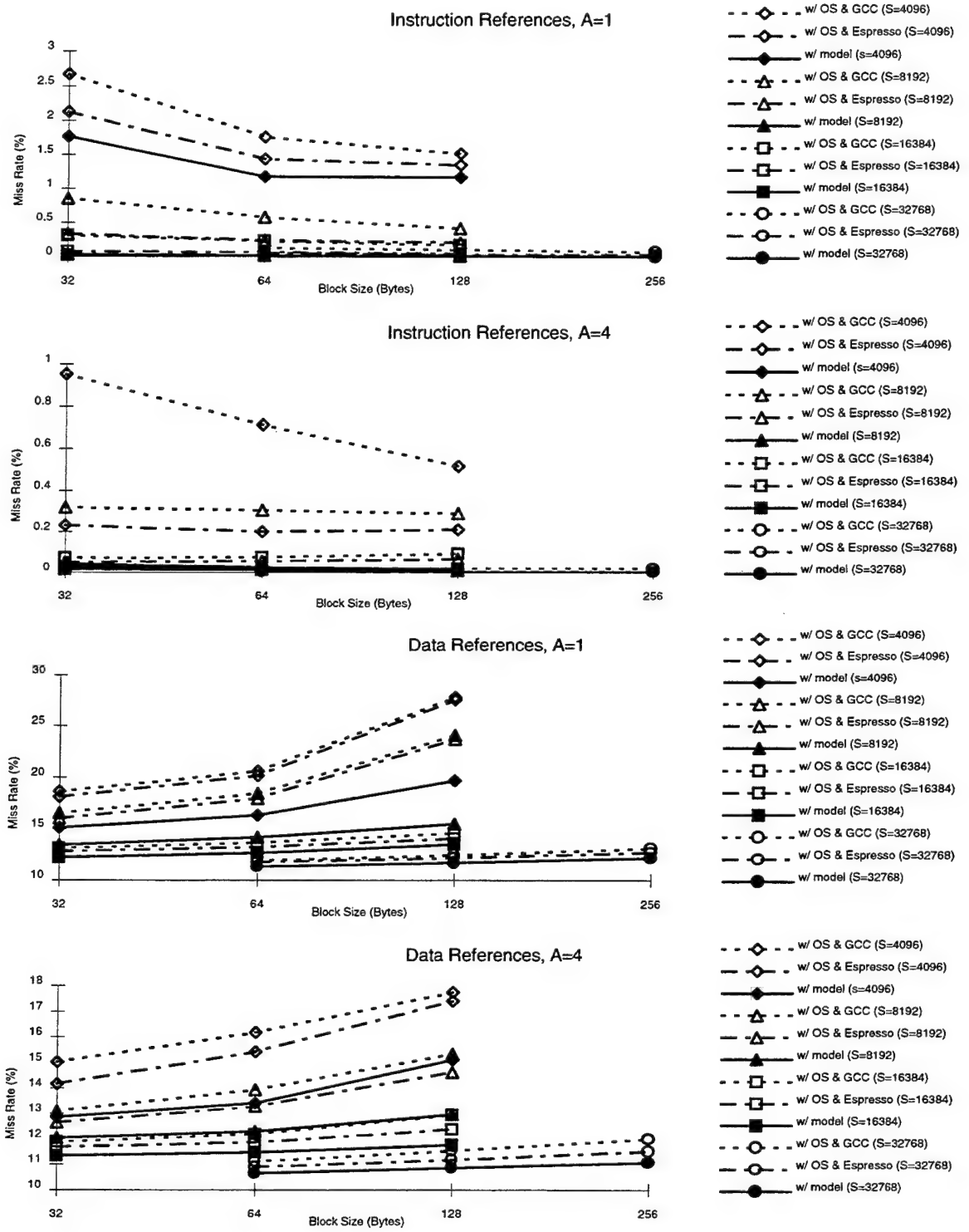

Figure 42: Model Results for Compress; n=2

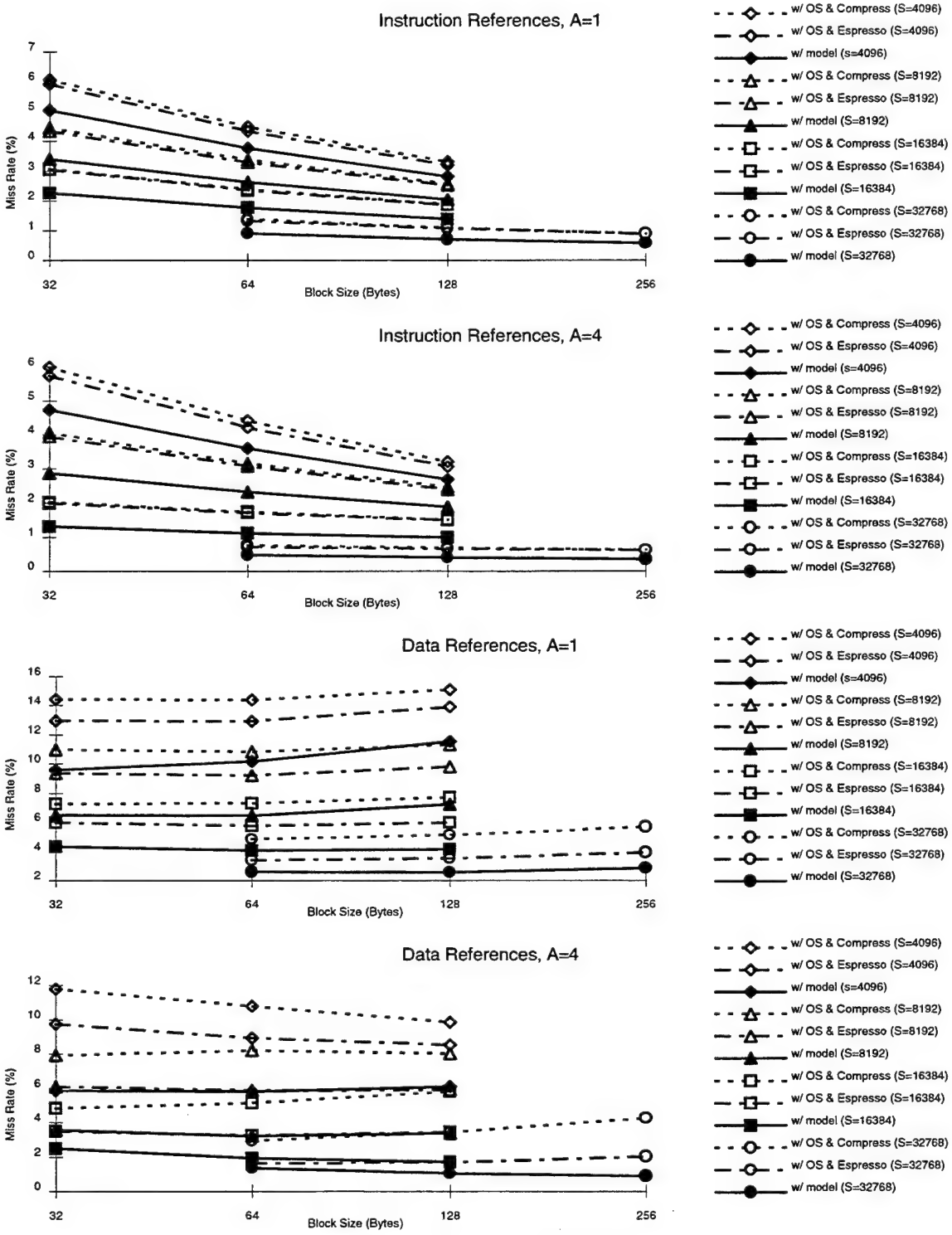

Figure 43: Model Results for GCC; n=2

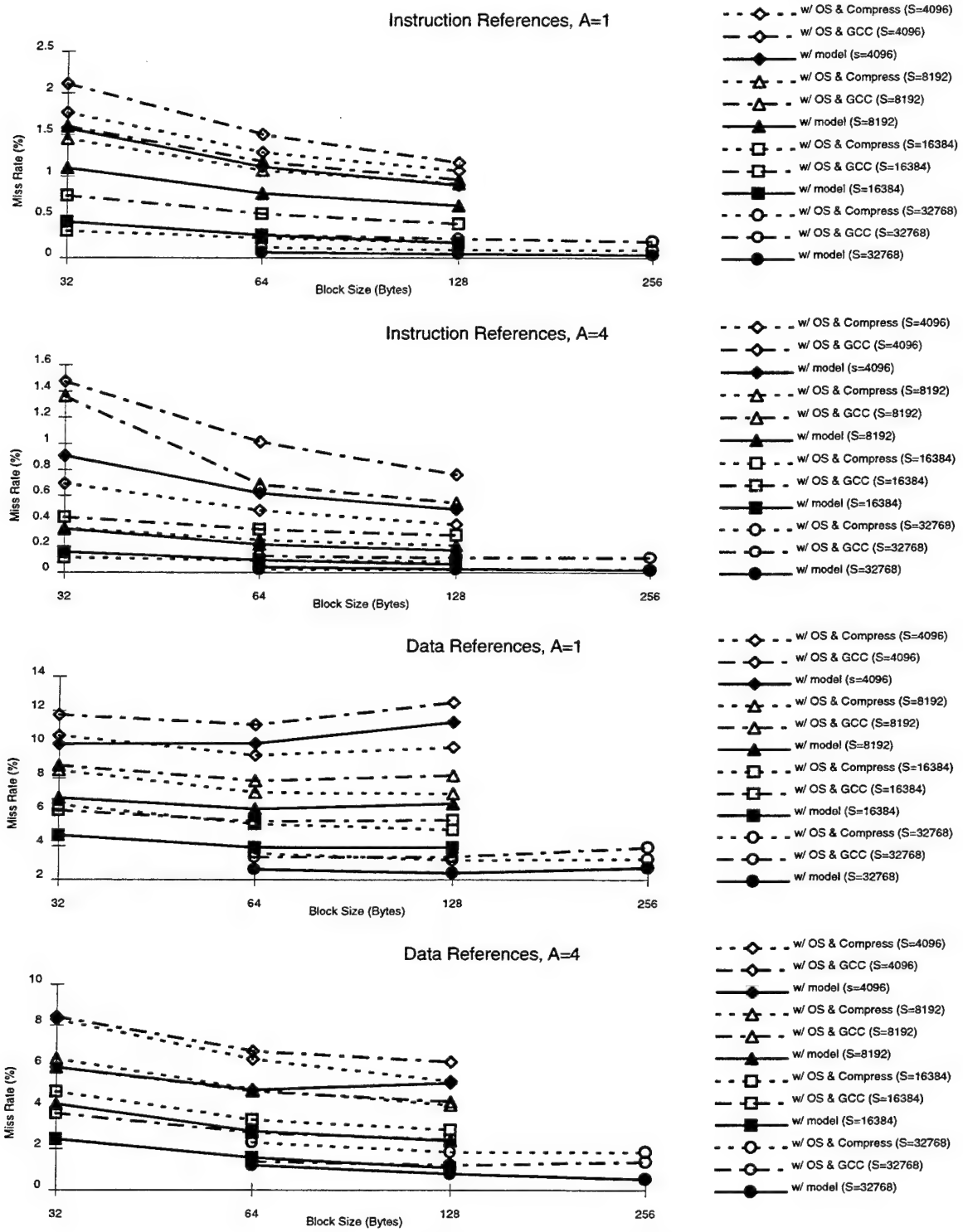

Figure 44: Model Results for Espresso; n=2

overwrote a process' own data ($\approx$ intrinsic interference), as opposed to overwriting another processes data ($\approx$ extrinsic interference). These plots are shown for each of the seven test cases in Figures 45 through 51.

As can be seen, the model underestimates the amount of extrinsic interference present in a multitasked situation. With a second program in the model, the primary source of interference is still intrinsic, as seen by the percentage of self overwrites, which, based on the previous results, is inaccurate. The only instances the model is even remotely correct is for the largest caches for GCC and Espresso.

Given the fact that the operating system is modeled fairly accurately, but the impact for other programs is not, the most likely source of error is in the impact to the cache at each context switch. The switch frequency is assumed to be more accurate. This is also supported by the assumptions used to develop the model. The most likely source of error is the footprint characterization. Using a simple function of the interruption interval is obviously an oversimplification. A more accurate model could be developed by using a more flexible model of footprint size and composition based on program features.

7.4 Summary

Based on the above results, the model described in section 6 does not adequately introduce the impact of context switches into a single process simulation. The interference generated approaches the level caused by the operating system, but is not significant enough to represent additional user programs. Given the assumptions used to develop the model, the most likely source of error is in the realization of context switch impact, in particular the computation of the program footprint. The method used was overly simplified, especially the relationship between block size and program footprint.

The difficulty of developing an accurate context switch model highlights the complexity of the cache environment. Cache performance is an intricate subject, and some aspects are not well understood. Analytical models can facilitate evaluation, but at the expense of accuracy. Any model will have to find a balance between these two goals. The requirement for accuracy reaffirms the need for analysis tools as described earlier, despite their own limitations.

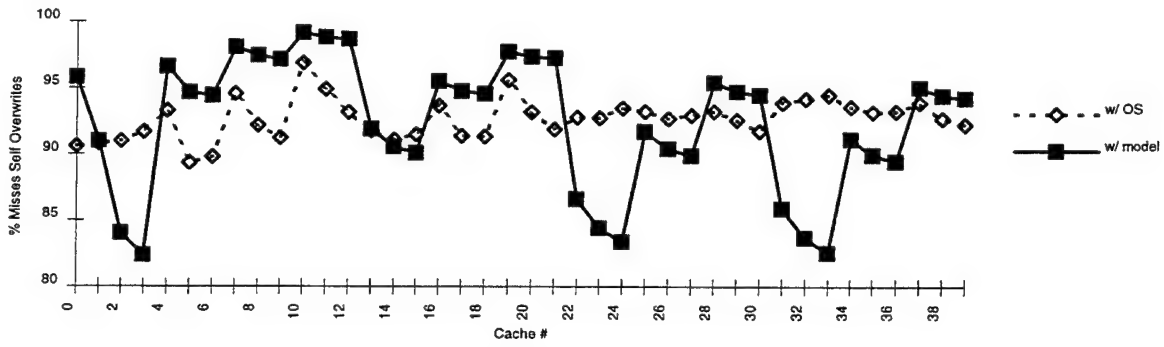

Figure 45: Percent Self Overwritten for Compress; n=1

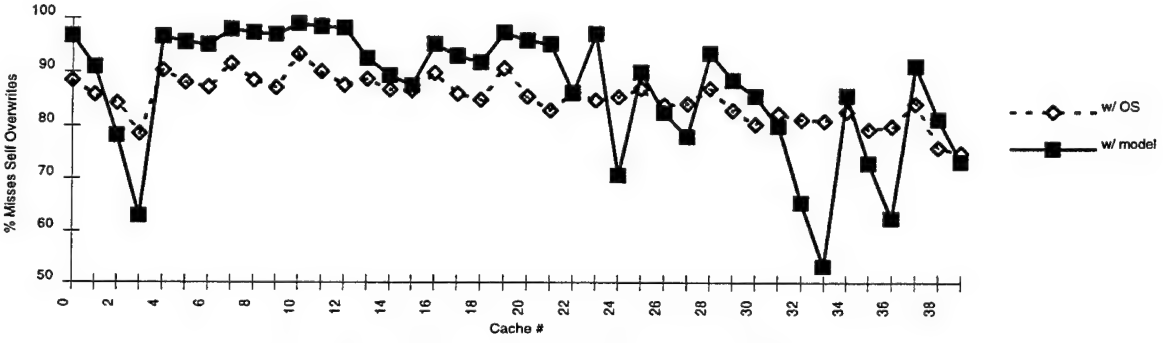

Figure 46: Percent Self Overwritten for GCC; n=1

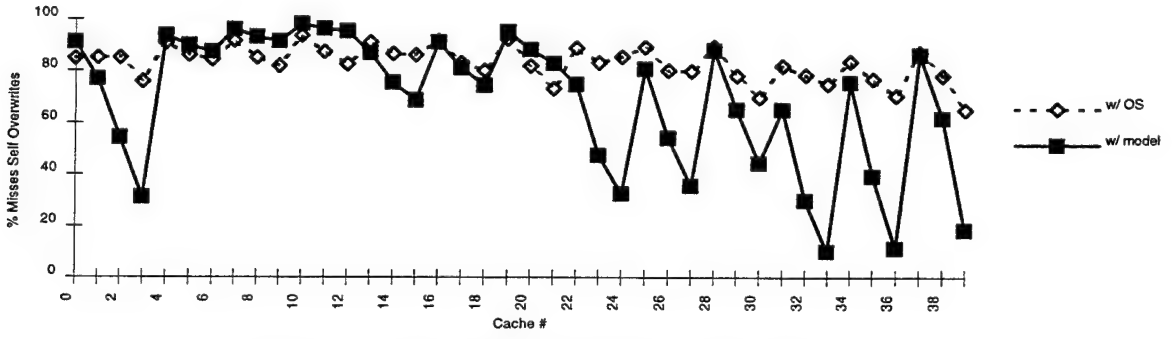

Figure 47: Percent Self Overwritten for Espresso; n=1

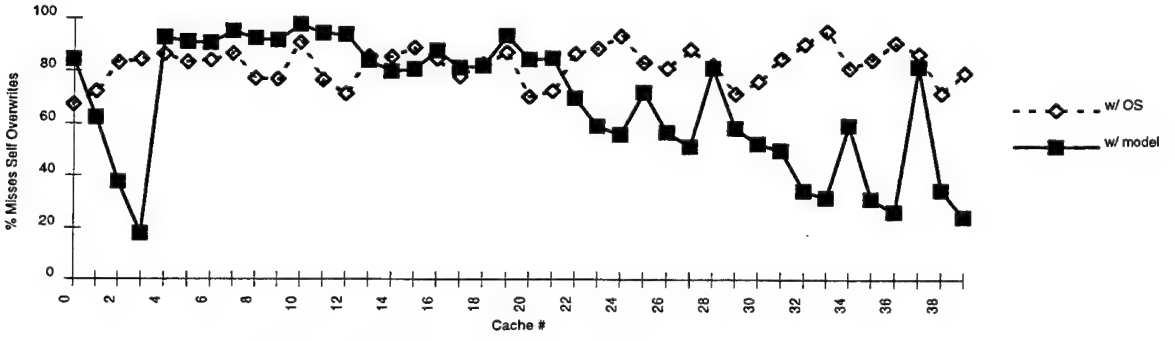

Figure 48: Percent Self Overwritten for Alvinn; n=1

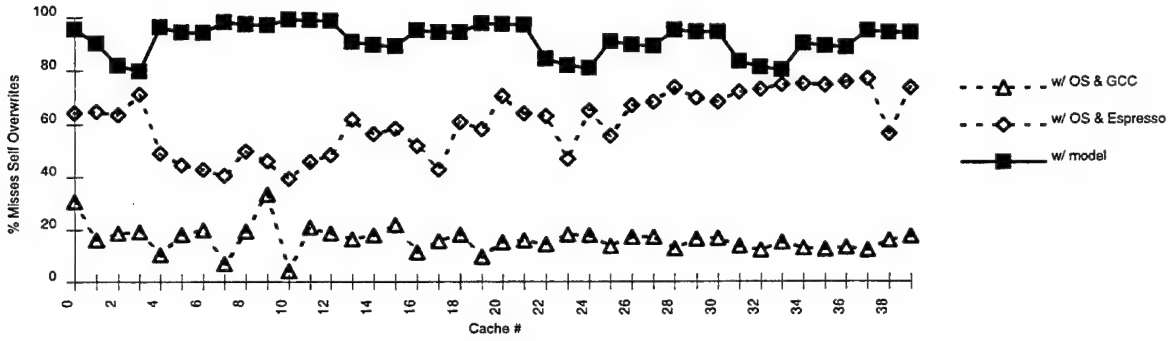

Figure 49: Percent Self Overwritten for Compress; n=2

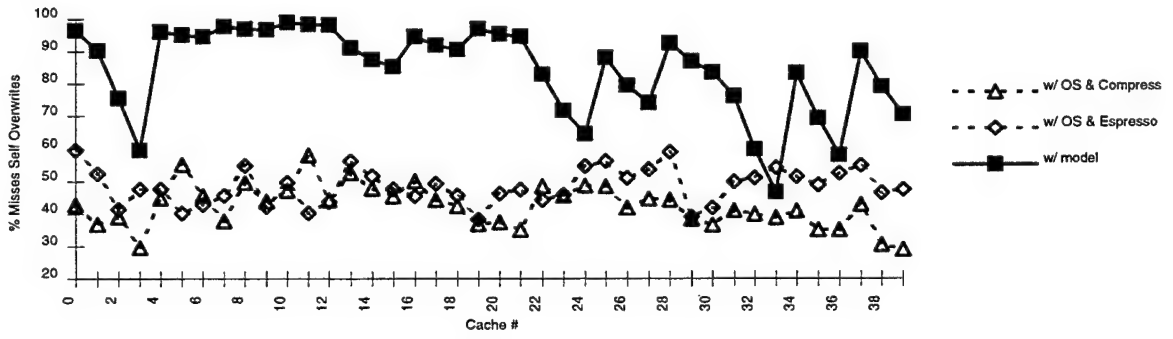

Figure 50: Percent Self Overwritten for GCC; n=2

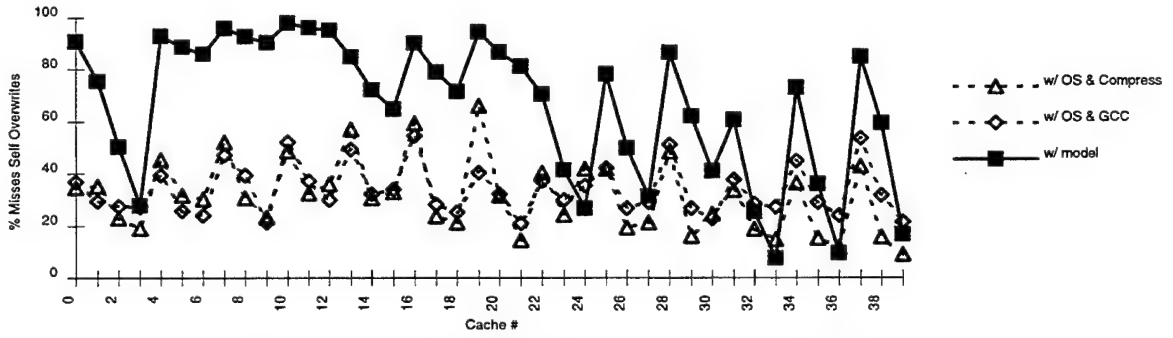

Figure 51: Percent Self Overwritten for Espresso; n=2

7.5 Future Work

While the model was not particularly successful in predicting interference, it does provide a theoretical foundation for further exploration. As discussed above, the primary limitation is the simplistic treatment of process footprints. Were this to be resolved and the footprints consider both the program in question and the cache block size, the model should perform much better.

Other potential improvements are a more detailed characterization of the operating system, to include its various composite threads. Also, the footprint of the operating system processes must be considered differently than user programs, due to their unique nature. The execution interval function can also be improved, by including specific program characteristics such as the frequency of system calls and interrupts generated by that particular program. Finally, additional aspects of the various existing analytical models can be incorporated to further simplify the simulations. A better understanding of the execution environment will allow more realistic assumptions to be used in that case.

8 Conclusions

The primary thrust of this research was the development and refinement of the ATOM based simulation capability for a complex workload. This was accomplished through the development of a very flexible and robust analysis program. This program is based on standard simulation tools, but incorporates novel techniques to allow a more comprehensive analysis. Partially based on the current work of others, many of these techniques still required extensive test and adaptation before their performance was adequate. Other areas, such as re-entrant analysis, were totally original. Several avenues of future work have also been highlighted, based on developing this work into an even more mature tool.

The cache simulations were performed as a demonstration of the overall potential of the simulation capability, as well as reinforcing assumptions about cache performance with operating system overhead and in the multiprocess environment. The context switch model attempted to combine both empirical and theoretical understanding of caches, and the testing portrayed a specific application of the ATOM tools created. These results were generally consistent with past endeavors, although highlighted some possible deficiencies in current methods and assumptions. The execution environment is quite complex, and aspects of its behavior are not particularly well understood. The ATOM tool promises to be a very effective and flexible tool for robust computer architecture analysis, however further work is necessary to fully realize its potential.

In the final analysis, the consideration of cache miss rates must be weighed with the impact of those miss rates on overall memory system performance. The actual goal of a cache is to improve memory access times. A cache with a very low miss rate but with a slow access time is just as much a problem as a cache with a high miss rate but very fast access time. Traffic between the various levels of the memory hierarchy will also play a factor, as the time to service a miss is also important. Other factors such as the area and power required for the cache must also be considered for an accurate appraisal of the cost and benefits of incorporating a certain cache design into a system. This work has been the first step towards such appraisals which include a comprehensive workload.

9 Contributions of this Thesis

- The majority of the work described in this thesis has revolved around developing the ATOM tracing capability for the operating system and multiple user programs. Previous work in this particular area is almost non-existent. ATOM itself is a well defined tool, but this type of implementation has not been studied before. A general method to instrument the kernel is outlined by Eustace and Chen in [20], but not well explored. Their material was used as a foundation, but expanded upon to develop the next generation of tools. The testing and refinement performed over the past year have made advances in several areas:
  - The cache simulation tools developed are much more comprehensive than any existing ATOM programs, providing more flexibility and detailed results.
  - The techniques proposed by Eustace and Chen have been extended to include not only the operating system but multiple user programs.
  - The issue of re-entrant analysis functions was explored for the first time. This will play a critical role in the exploration of certain applications such as the operating system.
  - Other limitations associated with using ATOM on the kernel are now more fully understood. Some were addressed in this work, while others will require further study to be completely resolved.
- The cache simulations served as a validation of the tools developed. The results confirmed the necessity for this type of work, revealing the significance of multiprogramming in workloads. The data gathered has affirmed theories about cache performance, and can be used to design more efficient memory caches.
- The context switch model attempts to combine both theoretical and empirical cache studies in an effort to achieve a balance between simplicity and accuracy. It is an extension of the basic cache model which synthetically generates the impact of multiprogramming. While not entirely successful, the testing does highlight gaps in current understanding of cache performance in a complex environment. This will serve as a background for more appropriate models, which should successfully reduce simulation processing.
- The most significant aspects of this thesis are the potential contributions to future work. With the capability developed here, a wide variety of additional cache studies are possible. With

some relatively minor modification, the tools developed can be adapted to a wide variety of program analyses. Most importantly, this work will provide the foundation to allow these studies to include the operating system, a subject that has not be well addressed in the past.

10 Acknowledgments

I would like to first thank my advisor, David Kaeli, for his guidance and motivation in completing this project. I would also like to thank Bradley Chen, Alan Eustace, and Greg Lueck for their considerable assistance in working with ATOM, and Liz Stewart for administration of the testbed system. Finally, I would like to thank Kristi Forbes for her invaluable moral support.

This thesis was funded by The Charles Stark Draper Laboratory through a Draper Fellowship under IR&D number 713, Fault Tolerant Computing. Publication of this thesis does not constitute approval by Draper of the findings or conclusions contained herein. It is published for the exchange and stimulation of ideas.

11 Bibliography

References

[1] A. Agarwal, *Analysis of Cache Performance for Operating Systems and Multiprogramming*, Kluwer,1989.

[2] A. Agarwal, J. Hennessey, and M. Horowitz, "Cache performance of Operating System and Multiprogramming Workloads", *ACM Transactions on Computer Systems*, Vol. 6 No. 4, Nov 88, pp. 393-431.

[3] A. Agarwal, M. Horowitz, and J. Hennessey, "An Analytical Cache Model", *ACM Transactions on Computer Systems*, Vol. 7 No. 2, May 89, pp. 184-215.

[4] E. Appleton, "DEC OSF/1: A Taste for Business", *The DEC Professional*, Vol. 13 No. 1, Jan 94, pp. 40-44.

[5] P. Argade, D. Charles, and C. Taylor, "A Technique for Monitoring Run Time Dynamics of an Operating System and a Microprocessor Executing User Applications", *ACM SIGPLAN Notices*, Vol. 29 No. 11, Nov 94, pp. 122-131.

[6] D. Bernstein, S. Gal, and M. Rodeh, "Mathematical Analysis of Statistical Sampling for Estimating Computer Cache Performance", *Communications In Statistics*, Vol. 12 No. 1, 1996, pp. 67-75.

[7] B. Bershad and B. Chen, "Avoiding Conflict Misses Dynamically in Large Direct Mapped Caches", *ACM SIGPLAN Notices*, Vol. 29 No. 11, Nov 94, pp. 158-170.

[8] A. Borg, R. Kessler, and D. Wall, "Generation and Analysis of Very Long Address Traces", *Computer Architecture News*, Vol. 18 No. 2, Jun 90, pp. 270-279.

[9] P. Bourne, "UNIX: More on DEC OSF/1 Migration", *The DEC Professional*, Vol. 13 No. 1, Jan 94, pp. 49-50.

[10] B. Chen, Assembly code provided in personal correspondence via email, Apr 12, 1996.

[11] B. Chen, *The Impact of Software Structure and Policy on CPU and Memory System Performance*, PhD Thesis Carnegie Mellon # CMU-CS-94-145, 1994.

[12] B. Chen and B. Bershad, "The Impact of Operating System Structure on Memory System Performance", *Operating Systems Review*, Vol. 27 No. 5, Dec 93, pp.120-133.

[13] B. Chen, D. Wall, and A. Borg, "Software Methods for System Address Tracing: Implementation and Validation", DEC WRL Research Report 94/6, 1994.

[14] T. Chen and J. Baer, "A Performance Study of Software and Hardware Data Prefetching Schemes", *Computer Architecture News*, Vol. 22 No. 2, Jun 94, pp. 223-232.

[15] F. Dahlgren, M. Dubois, and P. Stenstrom, "Combined Performance Gains of Simple Cache Extensions", *Computer Architecture News*, Vol. 22 No. 2, Jun 94, pp. 187-197.

[16] J. Denham, P. Long, and J. Woodward, "DEC OSF/1 Version 3.0 Symmetric Multiprocessing Implementation", *Digital Technical Journal*, Vol. 6 No. 3, Sum 94, pp. 29-43.

[17] T. Dutton, D. Eiref, H. Kurth, J. Reisert, and R. Stewart, "The Design of the DEC 3000 AXP Systems, Two High Performance Workstations", *Digital Technical Journal*, Vol. 4 No. 4, 92 spec, pp. 67-81.

[18] J. Dwyer and J. Richman, "OSF/1", *UNIX Review*, Vol. 10 No. 4, Apr 92, pp. 29-47.

[19] H. El-Rewini, H. Ali and T. Lewis, "Task Scheduling in Multiprocessing Systems", *Computer*, Vol. 28 No. 12, Dec 95, pp. 27-37.

[20] A. Eustace and B. Chen, "ATOM Kernel Instrumentation Guide Version 0.4 ", unpublished, Sep 1995.

[21] M. Evers, P. Chang, and Y. Patt, "Using Hybrid Predictors to Improve Branch Prediction Accuracy in the Presence of Context Switches", *Computer Architecture News*, Vol. 24 No. 2, Jun 96, pp. 3-11.

[22] J. Feldman and C. Retter, *Computer Architecture: A Designers Text Based on a Generic RISC*, McGraw Hill, 1994.

[23] J. Fraser, "Simple Modeling of Multiprocess Effects in Cache Simulations", unpublished, 1995.

[24] J. Fraser and D. Kaeli, "Operating System Impact on Cache Performance", unpublished, 1996.

[25] J. Gee, M. Hill, D. Pnevmatikatos, and A. Smith, "Cache Performance of the SPEC92 Benchmark Suite", *IEEE Micro*, Vol. 13 No. 4, Aug 93, pp. 17-27.

[26] M. Holliday and C. Ellis, "Accuracy of memory Reference Traces of Parallel Computations in Trace Driven Simulation", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 3 No. 1, Jan 92, pp. 97-109.

[27] G. Intrater and I. Spillinger, "Performance Evaluation of a Decoded Instruction Cache for Variable Instruction Length Computer", *IEEE Transactions on Computers*, Vol. 43 No. 10, Oct 94, pp. 1140-1150.

[28] Q. Jin and Y Sugasawa, "Representation and Analysis of Behavior for Multiprocess Systems by Using Stochastic Petri Nets", *Mathematical and Computer Modeling*, Vol. 22 No. 10-12, Nov-Dec 95, pp. 109-118.

[29] N. Jouppi, "Cache Write Policies and Performance", *Computer Architecture News*, Vol. 21 No. 2, Jun 93, pp. 191-201.

[30] K. Kavi, A Hurson, P. Patadia, E. Abraham, and P. Shanmugam, "Design of Cache Memories for Multithreaded Dataflow Architecture", *Computer Architecture News*, Vol. 23 No. 2, May 95, pp. 253-264.

[31] M. Kobayashi, "A Cache Multitasking Model", *Performance Evaluation Review*, Vol. 20 No. 2, Nov 92, pp. 27-37.

[32] J. Kuntz, "Performance Evaluation of Cache Architectures in Tightly Coupled Multiprocessor Systems", *Future Generations Computer Systems*, Vol. 10 No. 1, Oct 94, pp. 15-27.

[33] S. Laha, J. Patel, and R. Iyer, "Accurate Low-Cost Methods for Performance Evaluation of Cache memory Systems", *IEEE Transactions on Computers*, Vol. 37 No. 11, Nov 88, pp. 1325-1335.

[34] A. Lebeck and D. Wood, "Cache Profiling and the SPEC Benchmarks: A Case Study", *Computer*, Vol. 27 No. 10, Oct 94, pp. 15-26.

[35] S. Mahmud, "Comments on 'Synthetic Traces for Trace Driven Simulation of Cache Memories"', *IEEE Transactions on Computers*, Vol. 43 No. 1, Jan 94, pp. 125-126.

[36] M. Markowitz, "Cache Design", *EDN*, Vol. 36 No. 9, Apr 91, pp. 136-148.

[37] A. Maynard, C. Donnelly, and B. Olszewski, "Contrasting Characteristics and Cache Performance of Technical and Multi-User Commercial Workloads", *ACM SIGPLAN Notices*, Vol. 29 No. 11, Nov 94, pp. 145-156.

[38] D. McCrackin and S. Srinivasan, "Trace Driven Pipeline and Cache Simulation of Multithreaded Computers", *Simulation*, Vol. 63 No. 2, Aug 94, pp. 75-82.

[39] E. McLellan, "The Alpha AXP Architecture and 21064 Processor", *IEEE Micro*, Vol. 13 No. 3, Jun 93, pp. 36-47.

[40] E. McRae, "Benchmarking Real Time Operating Systems", *Dr Dobbs Journal*, Vol. 21 No. 5, May 96, pp. 48-58.

[41] J. Mogul and A. Borg, "The Effect of Context Switches on Cache Performance", *ACM SIGPLAN Notices*, Vol. 26 No. 4, Apr 91, pp. 75-84.

[42] D. Nicol and E. Carr, "Empirical Study of Parallel Trace Driven LRU Cache Simulators", *Simulation Digest*, Vol. 25 No. 1, Jul 95, pp. 166-169.

[43] D. Nicol, A. Greenberg, and B. Lubachevsky, "Massively Parallel Algorithms for Trace Driven Cache Simulations", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 5 No. 8, Aug 94, pp. 849-858.

[44] S. Oualline, *Practical C Programming*, O'Reilly and Associates, 1991.

[45] D. Pnevmatikatos and M. Hill, "Cache Performance of the Integer SPEC Benchmarks on a RISC", *Computer Architecture News*, Vol. 18 No. 2, Jun 1990, pp. 53-68.

[46] C. Prete, G. Prina, and L. Ricciardi, "A Trace Driven Simulator for Performance Evaluation of Cache Based Multiprocessor Systems", *IEEE Transactions on Parallel and Distributed Systems*, Vol. 6 No. 9, Sep 95, pp. 915-929.

[47] S. Przybylski, M. Horowitz, and J. Hennessey, "Performance Tradeoffs in Cache Design", *Computer Architecture News*, Vol. 16 No. 3, Jun 88, pp. 290-298.

[48] R. Quong, "Expected I Cache Miss Rates via the Gap Model", *Computer Architecture News*, Vol. 22 No. 2, Apr 94, pp. 372-383.

[49] R. Saavedra and A. Smith, "Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes", *IEEE Transactions on Computers*, Vol. 22 No. 10, Oct 95, pp. 1223-1235.

[50] D. Spinellis, "Trace: A Tools for Logging Operating System Call Transactions", *Operating System Review*, Vol. 28 no 4, Oct 94, pp. 56-62.

[51] A. Srivastava and A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools", *ACM SIGPLAN Notices*, Vol. 29 No. 6, Jun 94, pp. 196-205.

[52] W. Stallings, *Computer Organization and Architecture: Principles of Structure and Function*, Macmillan, 1990.

[53] H. Stark and J. Woods, *Probability, Random Processes, and Estimation Theory for Engineers*, Prentice Hall, 1994

[54] C. Stunkel and K. Fuchs, "TRAPEDS: Producing Traces for Multicomputers Via Execution Driven Simulation", *Performance Evaluation Review*, Vol. 17 No. 1, May 89, pp. 70-78.

[55] O. Temam, C. Fricker, and W. Jalby, "Cache Interference Phenomena", *Performance Evaluation Review*, Vol. 22 No. 1, May 94, pp. 261-271.

[56] D. Thiebaut and H. Stone, "Footprints in the Cache", *ACM Transactions on Computer Systems*, Vol. 5 No. 4, Nov 87, pp. 305-329.

[57] D. Thiebaut, J. Wolf, and H. Stone, "Synthetic Traces for Trace Driven Simulation of Cache Memories", *IEEE Transactions on Computers*, Vol. 41 No. 4, Apr 92, pp. 388-410.

[58] D. Thiebaut, J. Wolf, and H. Stone, "Corrigendum to 'Synthetic Traces for Trace Driven Simulation of Cache Memories"', *IEEE Transactions on Computers*, Vol. 42 No. 5, May 93, pp. 635-636.

[59] J. Torrellas, A. Gupta, and J. Hennessy, "Characterizing the Caching and Synchronization Performance of a Multiprocessor Operating System", *ACM SIGPLAN Notices*, Vol. 27 No. 9, Sep 92, pp. 162-174.

[60] R. Uhlig and T. Mudge, "Trace Driven Memory Simulation: A Survey", unpublished, 1996.

[61] W. Wang and J. Baer, "Efficient Trace Driven Simulation Methods for Cache Performance Analysis", *ACM Transactions on Computer Systems*, Vol. 9 No. 3, Aug 91, pp. 27-36.

[62] D. Whalley, "Fast Instruction Cache Performance Evaluation Using Compile Time Analysis", *Performance Evaluation Review*, Vol. 20 No. 1, Jun 92, pp. 13-22.

[63] Y. Wong and S Hwang, "Prediction of Memory Consumption in Conservative Parallel Simulation", *Simulation Digest*, Vol. 25 No. 1, Jul 95, pp. 199-202.

[64] E. Wu, Y. Hsu, and Y. Liu, "Efficient Stack Simulation for Set Associative Virtual Address Caches With Real Tags", *IEEE Transactions on Computers*, Vol. 44 No. 5, May 95, pp. 719-723.

[65] *Alpha AXP Architecture Handbook*, Digital Equipment Corporation, 1994.

[66] *ATOM Reference Manual*, Digital Equipment Corporation, 1993.

[67] *ATOM User Manual*, Digital Equipment Corporation, 1994.

[68] *ATOM User Manual*, Digital Equipment Corporation, 1995.

[69] *DEC 3000 Model 300 Series AXP Hardware Reference Guide*, Digital Equipment Corporation, 1994.

[70] *DEC OSF/1 Installation Guide*, Digital Equipment Corporation, 1994.

[71] *DEC OSF/1 Guide To Real-time Programming*, Digital Equipment Corporation, 1994.

[72] *DEC OSF/1 Technical Overview*, Digital Equipment Corporation, 1994.

[73] *Program Analysis Using Atom Tools*, Digital Equipment Corporation, 1996.

[74] On line documentation (SPEC92, ATOM, Dinero).

A Program Source Code

Programs are based primarily on the structure developed in [20] and past work from [23, 24]. Other sources for information include [44, 66, 67, 68, 73, 74]. The input and output file formats are shown first with short examples, followed by the various files and programs used. They are provided as a reference for future efforts as well as to help understanding of the material:

1. Input Format and Example
2. Output Format and Example
3. Cache Model Library
4. Kernel Instrumentation File
5. Kernel Analysis File
6. Program Instrumentation File
7. Program Analysis File
8. Sample Tool Description File
9. Context Switch Model Library
10. Model Analysis File

A.1 Input Format

The input file must be called `cache.in` and has the format:

- (simulation name)
- (number of processes in simulation)
- (name of each process (n-1 names, process 0 is assumed to be the OS)

  ⋮
- (number of caches in simulation)
- (cache definitions)

  ⋮

Names can contain up to 80 characters. Cache definitions consist of two lines. The first is a 0 or 1 denoting the cache type. The second contains the cache parameters in the forms shown below based on cache type:

Unified(0) (U cache size) (U block size) (U associativity)

Split(1) (I cache size) (I block size) (I associativity) (D cache size) (D block size) (D associativity)

An short example input file is shown below:

```
multi process test
3
cc1 -O -quiet stmt.i -o stmt
espresso tial.in > /dev/null
3
0
16384 64 2
1
16384 128 4 16384 128 4
1
32768 256 1 32768 256 1
```

A.2 Output Format

The simulation results were dumped to a file called cache.out. The output format has a banner page followed by a page of results for each cache. Results are recorded at the end of each program in the simulation, however the second set of data was removed from the example for brevity. The format is self evident from the example shown below. In hindsight, the output file should have used a format directly readable by a spreadsheet program. The format below is easy to understand, however it also requires manual entry of data into spreadsheets for analysis.

```
<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>
SIMULATION: multi process test
<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>

Number Tasks = 3

       #0: kernel

       #1: cc1 -O -quiet stmt.i -o stmt

       #2: espresso tial.in > /dev/null

Number Caches = 3
       (type, icsize, ilsize, iassoc, dcsize, dlsize, dassoc)

       #0: 0   16384    64   2

       #1: 1   16384   128   4   16384   128   4

       #2: 1   32768   256   1   32768   256   1


DATA AT END OF PROCESS 1
<><><><><><><><><><><><><><><><><><><><><><><><><>
simulation: multi process test
            (data at end of process 1)
-------------------------------------------------
CACHE # 0
cache type: 0 (0=unified, 1=split)
icache size: 16384
icache line size: 64
icache associativity: 2
     **********
     Process #0
         Inst      39004710 Miss      2739339 Perc 7.023098
         Data      16350661 Miss      3071643 Perc 18.786048
          read     10758087 Miss      2366717 Perc 21.999422
          writ      5592574 Miss       704926 Perc 12.604679
         TOTAL     55355371 Miss      5810982 Perc 10.497594
          Interferance (number times process 0 overwrote:)
               Process 0 =      2614797
               Process 1 =      2207422
```

```
            Process 2 =       988510
            Process 3 =          253
           (process 3 is invalid data)
     **********
     Process #1
         Inst     160240175 Miss      5166542 Perc 3.224249
         Data      69272178 Miss      4512864 Perc 6.514685
          read     50197333 Miss      3475694 Perc 6.924061
          writ     19074845 Miss      1037170 Perc 5.437371
         TOTAL    229512353 Miss      9679406 Perc 4.217379
          Interferance (number times process 1 overwrote:)
               Process 0 =      2175838
               Process 1 =      4910549
               Process 2 =      2287801
               Process 3 =            3
              (process 3 is invalid data)
     **********
     Process #2
         Inst     224015943 Miss      1813316 Perc 0.809458
         Data      63229661 Miss      3257726 Perc 5.152212
          read     51131731 Miss      2778587 Perc 5.434174
          writ     12097930 Miss       479139 Perc 3.960504
         TOTAL    287245604 Miss      5071042 Perc 1.765403
          Interferance (number times process 2 overwrote:)
               Process 0 =      1020129
               Process 1 =      2561443
               Process 2 =      1489470
               Process 3 =            0
              (process 3 is invalid data)
     ********************
     TOTAL FOR CACHE
         Inst     423260828 Miss      9719197 Perc 2.296267
         Data     148852500 Miss     10842233 Perc 7.283877
          read    112087151 Miss      8620998 Perc 7.691335
          writ     36765349 Miss      2221235 Perc 6.041654
         TOTAL    572113328 Miss     20561430 Perc 3.593944
simulation: multi process test
            (data at end of process 1)
-------------------------------------------------
CACHE # 1
cache type: 1 (0=unified, 1=split)
icache size: 16384
icache line size: 128
icache associativity: 4
dcache size: 16384
dcache line size: 128
dcache associativity: 4
     **********
     Process #0
         Inst      39028217 Miss      1297351 Perc 3.324136
         Data      16360315 Miss      2091714 Perc 12.785292
```

```
        read     10764480 Miss      1706268 Perc 15.850910
        writ      5595835 Miss       385446 Perc 6.888087
       TOTAL     55388532 Miss      3389065 Perc 6.118712
        Interferance (number times process 0 overwrote:)
             Process 0 =      1358317
             Process 1 =      1358773
             Process 2 =       671722
             Process 3 =          253
            (process 3 is invalid data)
    **********
    Process #1
        Inst     160240175 Miss      2378836 Perc 1.484544
        Data      69272178 Miss      2370733 Perc 3.422345
        read      50197333 Miss      1965331 Perc 3.915210
        writ      19074845 Miss       405402 Perc 2.125323
       TOTAL    229512353 Miss      4749569 Perc 2.069418
        Interferance (number times process 1 overwrote:)
             Process 0 =      1356440
             Process 1 =      2358083
             Process 2 =      1945071
             Process 3 =            3
            (process 3 is invalid data)
    **********
    Process #2
        Inst     224033574 Miss       652803 Perc 0.291386
        Data      63235212 Miss      1542671 Perc 2.439576
        read      51136035 Miss      1321124 Perc 2.583548
        writ      12099177 Miss       221547 Perc 1.831091
       TOTAL    287268786 Miss      2195474 Perc 0.764258
        Interferance (number times process 2 overwrote:)
             Process 0 =       674120
             Process 1 =       993262
             Process 2 =       488640
             Process 3 =            0
            (process 3 is invalid data)
    ********************
    TOTAL FOR CACHE
        Inst     423301966 Miss      4328990 Perc 1.022672
        Data     148867705 Miss      6005118 Perc 4.033862
        read     112097848 Miss      4992723 Perc 4.453897
        writ      36769857 Miss      1012395 Perc 2.753329
       TOTAL    572169671 Miss     10334108 Perc 1.806126
simulation: multi process test
          (data at end of process 1)
------------------------------------------------
CACHE # 2
cache type: 1 (0=unified, 1=split)
icache size: 32768
icache line size: 256
icache associativity: 1
dcache size: 32768
```

```
dcache line size: 256
dcache associativity: 1
     **********
     Process #0
         Inst      39100207 Miss       877237 Perc 2.243561
         Data      16384285 Miss      2191363 Perc 13.374786
          read     10780440 Miss      1793502 Perc 16.636631
          writ      5603845 Miss       397861 Perc 7.099786
         TOTAL     55484492 Miss      3068600 Perc 5.530554
          Interferance (number times process 0 overwrote:)
               Process 0 =      1704283
               Process 1 =       946851
               Process 2 =       417213
               Process 3 =          253
              (process 3 is invalid data)
     **********
     Process #1
         Inst     160240175 Miss      1414353 Perc 0.882646
         Data      69272178 Miss      2717362 Perc 3.922732
          read     50197333 Miss      2261685 Perc 4.505588
          writ     19074845 Miss       455677 Perc 2.388890
         TOTAL    229512353 Miss      4131715 Perc 1.800215
          Interferance (number times process 1 overwrote:)
               Process 0 =       942089
               Process 1 =      2260273
               Process 2 =       929350
               Process 3 =            3
              (process 3 is invalid data)
     **********
     Process #2
         Inst     224033574 Miss       435774 Perc 0.194513
         Data      63235212 Miss      2459827 Perc 3.889964
          read     51136035 Miss      2205351 Perc 4.312714
          writ     12099177 Miss       254476 Perc 2.103250
         TOTAL    287268786 Miss      2895601 Perc 1.007976
          Interferance (number times process 2 overwrote:)
               Process 0 =       422012
               Process 1 =       924590
               Process 2 =      1548999
               Process 3 =            0
              (process 3 is invalid data)
     ********************
     TOTAL FOR CACHE
         Inst     423373956 Miss      2727364 Perc 0.644197
         Data     148891675 Miss      7368552 Perc 4.948935
          read    112113808 Miss      6260538 Perc 5.584092
          writ     36777867 Miss      1108014 Perc 3.012720
         TOTAL    572265631 Miss     10095916 Perc 1.764201
DATA AT END OF PROCESS 2
<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>
(format repeats for data at end of second process)
```

A.3 Cache Model Library

The following file, cache.h, was used as a definition/procedure library for the basic cache simulator:

```
/* CACHE.H */
/* CACHE SIMULATION LIBRARY */
/* JOHN FRASER */

/* SIMULATION CHARACTERISTICS */
/* MAXIMUM NUMBER OF CACHES IN SIMULATION */
#define MAXCACHES 40
/* MAXIMUM NUMBER OF PROCESSES IN SIMULATION */
#define MAXTASKS 4
/* MAXIMUM NUMBER OF LINES (CSIZE/(BSIZE*ASSOC)) IN CACHES */
#define MAXLINE 512
/* MAXIMUM ASSOCIATIVITY OF CACHES */
#define MAXASSOC 4

/* CACHE PARAMETERS */
typedef struct
  {
  /* CACHE TYPE (0=UNIFIED, 1=SPLIT) */
  int type;
  /* CACHE SIZE FOR EACH SECTION (0=UNIFIED/INST, 1=DATA) */
  int csize[2];
  /* BLOCK SIZE FOR EACH SECTION */
  int bsize[2];
  /* ASSOCIATIVITY FOR EACH SECTION */
  int assoc[2];
  /* BIT SHIFT USED TO ISOLATE TAG FROM ADDRESS */
  int tshift[2];
  /* BIT SHIFT USED TO ISOLATE LINE FROM ADDRESS */
  int lshift[2];
  /* BIT MASK USED TO ISOLATE LINE FROM ADDRESS */
  int lmask[2];
  } param;

/* CACHE BLOCK STORAGE */
typedef struct
  {
  /* BLOCK TAG */
  long tag;
  /* BLOCK 'USE BITS' FOR ASSOCIATIVE CACHES */
  unsigned long use;
  /* BLOCK OWNER PROCESS */
  int task;
  } block;

/* CACHE PERFORMANCE STATISTICS */
typedef struct
  {
```

```
  /* NUMBER OF INSTRUCTION FETCHES */
  unsigned long instcnt;
  /* NUMBER OF DATA LOADS */
  unsigned long readcnt;
  /* NUMBER OF DATA STORES */
  unsigned long writcnt;
  /* NUMBER OF OVERWRITES OVER EACH PROCESS */
  /* NUMTASKS+1 = INVALID DATA */
  unsigned long interfere[MAXTASKS+1];
  /* NUMBER OF INSTRUCTION FETCH MISSES */
  unsigned long instmisscnt;
  /* NUMBER OF DATA LOAD MISSES */
  unsigned long readmisscnt;
  /* NUMBER OF DATA STORE MISSES */
  unsigned long writmisscnt;
  } stats;

/* STRING DEFINITION */
typedef char string[80];

/* SHARED ATOM DATA */
typedef struct
  {
  /* NUMBER OF CACHES IN USE */
  int numcaches;
  /* NUMBER OF CACHES IN SIMULAITON */
  int actcaches;
  /* NUMBER OF PROCESSES IN SIMULATION */
  int numtasks;
  /* NUMBER OF PROCESSES CURRENTLY EXECUTING */
  int count;
  /* PID OF CURRENT PROCESS */
  int curtask;
  /* PROCESS NAMES */
  string name[MAXTASKS];
  /* CACHE PARAMTERS */
  param para[MAXCACHES];
  /* CACHE STATE (BLOCK INFORMATION) */
  block data[MAXCACHES][2][MAXLINE][MAXASSOC];
  /* PERFORMANCE STATISTICS */
  stats stat[MAXCACHES][MAXTASKS];
  } datablock;

/* INTEGER LOG2 FUNCTION */
int mylog2(int num)
  {
  if (num < 2)
    return(0);
  else
    return(1 + mylog2(num/2));
  }
```

A.4 Kernel Instrumentation File

The kernel instrumentation file `kern.inst.c` is responsible for adding the calls to the analysis routines at the appropriate points. A call to the initialization function is made when the program is initially loaded, and thereafter at each data reference and sets of instructions, calls are made to the various analysis routines. A call is inserted at the start of each hardclock interrupt service routine for scaling purposes. Note the test to check for the kernel procedures which cannot be instrumented.

```
/* KERN.INST.C */
/* KERNEL INSTRUMENTATION FILE */
/* JOHN FRASER */

#include <string.h>
#include <cmplrs/atom.inst.h>

/* DEFINE PROCESS ID */
#define PROCNUM 0

/* TEST FOR ROUTINES WHICH CANNOT BE TRACED */
int CanInstrument(Proc *p)
  {
  const char* name = ProcFileName(p);
  return(strcmp("../../../../src/kernel/arch/alpha/locore.s",name)!=0 &&
         strcmp("../../../../src/kernel/arch/alpha/lockprim.s",name)!=0 &&
         strcmp("../../../../src/kernel/arch/alpha/spl.s",name)!=0);
  }

/* INSTRUMENT:                                */
/*      ALL DATA REFERENCES AND               */
/*      SETS OF 8 INSTRUCTIONS OR LESS        */
/*      (WITHIN SAME BASIC BLOCK)             */
/* ANALYSIS ROUTINES:                         */
/*      INSTRUCTION FETCH(ADDRESS,PID,NUMBER)*/
/*      DATA LOAD(ADDRESS,PID)                */
/*      DATA STORE(ADDRESS,PID)               */
unsigned InstrumentAll(int argc, char** argv)
  {
  Obj* o;
  Proc* p;
  Block* b;
  Inst* i;
  /* ADD PROCEDURE PROTOTYPES */
  AddCallProto("initcache()");
  AddCallProto("instref(REGV, int, int)");
  AddCallProto("readref(VALUE, int)");
  AddCallProto("writref(VALUE, int)");
  AddCallProto("skipcall(REGV, REGV)");
  /* ADD INITIALIZATION CALL */
  AddCallProgram(ProgramBefore,"initcache");
  /* ITERATE THROUGH ORIGINAL CODE ADDING REFERENCE CALLS */
  o = GetFirstObj();
  if (BuildObj(o)) return 1;
```

```
p = GetNamedProc("hardclock");
/* ADD CALL FOR HARDCLOCK SCALING */
AddCallProc(p, ProcBefore, "skipcall", REG_SP, REG_RA);
for (p=GetFirstObjProc(o); p!=NULL; p=GetNextProc(p))
  {
  if (CanInstrument(p))
    {
    for (b=GetFirstBlock(p); b!=NULL; b=GetNextBlock(b))
      {
      long pcEnd = InstPC(GetLastInst(b));
      int count = 0;
      for (i=GetFirstInst(b); i!=NULL; i=GetNextInst(i))
        {
        /* INSTRUCTION FETCH */
        if ((count & 7) == 0)
          {
          int instRem = ((pcEnd-InstPC(i))/4)+1;
          int instrLine = (instRem > 8) ? 8 : instRem;
          AddCallInst(i,InstBefore, "instref", REG_PC, PROCNUM, instrLine);
          }
        count++;
        /* DATA LOAD */
        if (IsInstType(i, InstTypeLoad))
          AddCallInst(i, InstBefore, "readref", EffAddrValue, PROCNUM);
        /* DATA STORE */
        if (IsInstType(i, InstTypeStore))
          AddCallInst(i, InstBefore, "writref", EffAddrValue, PROCNUM);
        }
      }
    }
  }
WriteObj(o);
return(0);
}
```

A.5 Kernel Analysis File

The kernel analysis file `kern.anal.c` defines the analysis routines called in the instrumentation file, and any other utility functions/procedures. There are 4 analysis routines to consider:

Initialization The initialization routine is responsible for establishing the basic simulation parameters when the kernel is loaded. The simulator is essentially put into a paused simulation state (0 caches) so that it is not actively capturing and processing references until a test program is started.

Hardclock Scaling This procedure will discard a certain number of hardclock interrupts controlled by a scaling factor.

Instruction Fetch Routine The instruction fetch routine is responsible for servicing instruction fetches in the reference stream. It processes each set of references in the cache based on the sets starting address, the number of instructions in the set, and the PID of the sending process. Using a PID allows the same code to be used for each process's analysis routines as well as maintaining cache coherency.

Data Load Routine The data load routine is responsible for servicing the data loads in the reference stream. It is almost identical to the previous routine except for the necessity of determining which cache to access depending on a split or unified model, and the fact that it services only a single reference at a time.

Data Store Routine The analysis routine for data stores, it is almost identical to the data load routine except for incrementing different counters.

The similarities between each routine would suggest that the common aspects be defined in a separate function which is called by each analysis routine, but this increases the processing latency by an unacceptable degree. The data used by these routines is defined in the library file and is implemented as global variables.

```
/* KERN.ANAL.C */
/* KERNEL ANALYSIS FILE */
/* JOHN FRASER */

/* HARDCLOCK SCALING VALUE */
#define SCALE 3

#include "cache.h"
#include <stdio.h>
#include <c_asm.h>

/* SHARED CACHE DATA */
datablock satom;
/* HARDCLOCK SCALING DATA */
int clockscale = 1;
int clockcount = 0;

/* INITIALIZE BASIC PARAMETERS */
/* SIMULATION (CAPTURE) DISABLED */
void initcache()
  {
  satom.numcaches = 0;
```

```
  satom.actcaches = 0;
  satom.numtasks = 0;
  satom.curtask = 0;
  satom.count = 0;
  clockscale = SCALE;
  clockcount = 0;
  return;
  }

/* HARDCLOCK SCALING */
void skipcall(unsigned long sp, unsigned long ra)
  {
  clockcount++;
  if (clockcount >= clockscale)
    {
    clockcount = 0;
    return;
    }
  asm("mov %a0, %sp",sp);
  asm("mov %a1, %ra",ra);
  asm("ret %zero, (%ra)");
  return;
  }

/* SCALING EMERGENCY */
void KernelPanic()
  {
  clockscale = 1;
  return;
  }

/* INSTRUCTION REFERENCE ROUTINE */
void instref(long addr, int proc, int count)
  {
  int x, leastx;
  unsigned long leastused;
  long aline, atag;
  int cnum, hit;
  /* PAUSE CAPTURE (RE-ENTRANCE) */
  int tempnumcaches = satom.numcaches;
  satom.numcaches = 0;
  /* PROCESS REFERENCES IN EACH CACHE */
  for (cnum=0; cnum<tempnumcaches; cnum++)
    {
    int assoc = (satom.para[cnum]).assoc[0];
    /* UPDATE STATISTICS */
    ((satom.stat[cnum][proc]).instcnt) += count;
    /* PARSE ADDRESS */
    aline = (addr & (satom.para[cnum]).lmask[0]) >>
                    (satom.para[cnum]).lshift[0];
    atag = addr >> (satom.para[cnum]).tshift[0];
```

```
    /* UPDATE 'USE BITS' AND CHECK FOR HIT */
    hit = 0;
    for (x=0; x<assoc; x++)
      {
      ((satom.data[cnum][0][aline][x]).use)++;
      if (((satom.data[cnum][0][aline][x]).tag == atag) &&
          ((satom.data[cnum][0][aline][x]).task == proc))
        {
        (satom.data[cnum][0][aline][x]).use = 0;
        hit = 1;
        }
      }
    /* IF NO HIT, FIND LRU BLOCK TO EVICT */
    if (hit == 0)
      {
      /* FIND LRU */
      leastused = 0;
      for (x=0; x<assoc; x++)
        {
        if (((satom.data[cnum][0][aline][x]).use >= leastused) ||
            ((satom.data[cnum][0][aline][x]).task ==
                                              satom.numtasks))
          {
          leastused = (satom.data[cnum][0][aline][x]).use;
          leastx = x;
          }
        if ((satom.data[cnum][0][aline][x]).task ==
                                             satom.numtasks)
          x = assoc;
        }
      /* UPDATE STATISTICS */
      ((satom.stat[cnum][proc]).instmisscnt)++;
      ((satom.stat[cnum][proc]).interfere[
       (satom.data[cnum][0][aline][leastx]).task])++;
      /* UPDATE CACHE DATA */
      (satom.data[cnum][0][aline][leastx]).tag = atag;
      (satom.data[cnum][0][aline][leastx]).use = 0;
      (satom.data[cnum][0][aline][leastx]).task = proc;
      }
    }
  /* RESUME CAPTURE */
  satom.numcaches = tempnumcaches;
  return;
  }

/* DATA LOAD ROUTINE */
void readref(long addr, int proc)
  {
  int index;
  int x, leastx;
  unsigned long leastused;
```

```
long aline, atag;
int cnum, hit;
/* PAUSE CAPTURE (RE-ENTRANCE) */
int tempnumcaches = satom.numcaches;
satom.numcaches = 0;
/* PROCESS REFERENCE IN EACH CACHE */
for (cnum=0; cnum<tempnumcaches; cnum++)
  {
  int type = (satom.para[cnum]).type;
  int assoc = (satom.para[cnum]).assoc[type];
  /* UPDATE STATISTICS */
  ((satom.stat[cnum][proc]).readcnt)++;
  /* PARSE ADDRESS */
  aline = (addr & (satom.para[cnum]).lmask[type]) >>
                  (satom.para[cnum]).lshift[type];
  atag = addr >> (satom.para[cnum]).tshift[type];
  /* UPDATE 'USE BITS' AND CHECK FOR HIT */
  hit = 0;
  for (x=0; x<assoc; x++)
    {
    ((satom.data[cnum][type][aline][x]).use)++;
    if (((satom.data[cnum][type][aline][x]).tag == atag) &&
        ((satom.data[cnum][type][aline][x]).task == proc))
      {
      (satom.data[cnum][type][aline][x]).use = 0;
      hit = 1;
      }
    }
  /* IF NO HIT, FIND LRU BLOCK TO EVICT */
  if (hit == 0)
    {
    /* FIND LRU */
    leastused = 0;
    for (x=0; x<assoc; x++)
      {
      if (((satom.data[cnum][type][aline][x]).use >= leastused) ||
          ((satom.data[cnum][type][aline][x]).task ==
                                              satom.numtasks))
        {
        leastused = (satom.data[cnum][type][aline][x]).use;
        leastx = x;
        }
      if ((satom.data[cnum][type][aline][x]).task ==
                                             satom.numtasks)
        x = assoc;
      }
    /* UPDATE STATISTICS */
    ((satom.stat[cnum][proc]).readmisscnt)++;
    ((satom.stat[cnum][proc]).interfere[
     (satom.data[cnum][type][aline][leastx]).task])++;
    /* UPDATE CACHE DATA */
```

```
      (satom.data[cnum][type][aline][leastx]).tag = atag;
      (satom.data[cnum][type][aline][leastx]).use = 0;
      (satom.data[cnum][type][aline][leastx]).task = proc;
      }
    }
  /* RESUME CAPTURE */
  satom.numcaches = tempnumcaches;
  return;
  }

/* DATA STORE ROUTINE */
void writref(long addr, int proc)
  {
  int index;
  int x, leastx;
  unsigned long leastused;
  long aline, atag;
  int cnum, hit;
  /* PAUSE CAPTURE (RE-ENTRANCE) */
  int tempnumcaches = satom.numcaches;
  satom.numcaches = 0;
  /* PROCESS REFERENCE IN EACH CACHE */
  for (cnum=0; cnum<tempnumcaches; cnum++)
    {
    int type = (satom.para[cnum]).type;
    int assoc = (satom.para[cnum]).assoc[type];
    /* UPDATE STATISTICS */
    ((satom.stat[cnum][proc]).writcnt)++;
    /* PARSE ADDRESS */
    aline = (addr & (satom.para[cnum]).lmask[type]) >>
                    (satom.para[cnum]).lshift[type];
    atag = addr >> (satom.para[cnum]).tshift[type];
    /* UPDATE 'USE BITS' AND CHECK FOR HIT */
    hit = 0;
    for (x=0; x<assoc; x++)
      {
      ((satom.data[cnum][type][aline][x]).use)++;
      if (((satom.data[cnum][type][aline][x]).tag == atag) &&
          ((satom.data[cnum][type][aline][x]).task == proc))
        {
        (satom.data[cnum][type][aline][x]).use = 0;
        hit = 1;
        }
      }
    /* IF NO HIT, FIND LRU BLOCK TO EVICT */
    if (hit == 0)
      {
      /* FIND LRU */
      leastused = 0;
      for (x=0; x<assoc; x++)
        {
```

```
      if (((satom.data[cnum][type][aline][x]).use >= leastused) ||
          ((satom.data[cnum][type][aline][x]).task == satom.numtasks))
        {
        leastused = (satom.data[cnum][type][aline][x]).use;
        leastx = x;
        }
      if ((satom.data[cnum][type][aline][x]).task == satom.numtasks)
        x = assoc;
      }
    /* UPDATE STATISTICS */
    ((satom.stat[cnum][proc]).writmisscnt)++;
    ((satom.stat[cnum][proc]).interfere[
     (satom.data[cnum][type][aline][leastx]).task])++;
    /* UPDATE CACHE DATA */
    (satom.data[cnum][type][aline][leastx]).tag = atag;
    (satom.data[cnum][type][aline][leastx]).use = 0;
    (satom.data[cnum][type][aline][leastx]).task = proc;
    }
  }
/* RESUME CAPTURE */
satom.numcaches = tempnumcaches;
return;
}
```

A.6 Program Instrumentation File

The program instrumentation file prog.inst.c is not substantially different from the kernel version. The primary change is the removal of the test for specific procedures which cannot be instrumented. The other alteration is the inclusion of a procedure at program end to write the simulations results to file. If multiple test programs are used, each uses a different instrumentation file with a unique process identifier assigned in the #define statement.

```
/* PROG.INST.C */
/* PROGRAM INSTRUMENTATION FILE */
/* JOHN FRASER */

#include <string.h>
#include <cmplrs/atom.inst.h>

/* DEFINE PROCESS ID */
#define PROCNUM 1

/* INSTRUMENT:                                */
/*      ALL DATA REFERENCES AND               */
/*      SETS OF 8 INSTRUCTIONS OR LESS        */
/*      (WITHIN SAME BASIC BLOCK)             */
/* ANALYSIS ROUTINES                          */
/*      INSTRUCTION FETCH(ADDRESS,PID,NUMBER)*/
/*      DATA LOAD(ADDRESS,PID)                */
/*      DATA STORE(ADDRESS,PID)               */
unsigned InstrumentAll(int argc, char** argv)
  {
  Obj* o;
  Proc* p;
  Block* b;
  Inst* i;
  /* ADD PROCEDURE PROTOTYPES */
  AddCallProto("initcache(int)");
  AddCallProto("instref(REGV, int, int)");
  AddCallProto("readref(VALUE, int)");
  AddCallProto("writref(VALUE, int)");
  AddCallProto("printres(int)");
  /* ADD INITIALIZATION CALL */
  AddCallProgram(ProgramBefore, "initcache", PROCNUM);
  /* ADD RESULTS OUTPUT CALL */
  AddCallProgram(ProgramAfter, "printres", PROCNUM);
  /* ITERATE THROUGH ORIGINAL CODE ADDING REFERENCE CALLS */
  o = GetFirstObj();
  if (BuildObj(o)) return 1;
  for (p=GetFirstObjProc(o); p!=NULL; p=GetNextProc(p))
    {
    for (b=GetFirstBlock(p); b!=NULL; b=GetNextBlock(b))
      {
      long pcEnd = InstPC(GetLastInst(b));
      int count = 0;
      for (i=GetFirstInst(b); i!=NULL; i=GetNextInst(i))
```

```
      {
      if ((count & 7) == 0)
        {
        int instRem = ((pcEnd-InstPC(i))/4)+1;
        int instrLine = (instRem > 8) ? 8 : instRem;
        AddCallInst(i,InstBefore, "instref", REG_PC, PROCNUM, instrLine);
        }
      count++;
      if (IsInstType(i, InstTypeLoad))
        AddCallInst(i, InstBefore, "readref", EffAddrValue, PROCNUM);
      if (IsInstType(i, InstTypeStore))
        AddCallInst(i, InstBefore, "writref", EffAddrValue, PROCNUM);
      }
    }
  }
WriteObj(o);
return(0);
}
```

A.7 Program Analysis File

The program analysis file `prog.anal.c` is almost identical to the kernel version, except for the initialization and conclusion routines. The reference processing routines perform the same function, the other two are described below:

Initialization The initialization routine is much more complex than its kernel equivalent. First it must map the shared data into the program's address space via the `/dev/mmap` utility. If the test program is the first to be executed for that simulation, it also reads the simulation data from the input file, initializes the cache data, and enables the simulation.

Conclusion The final routine is not present in the kernel because it is executed at program completion. It is responsible for writing the simulation results to the output file.

```
/* PROG.ANAL.C */
/* PROGRAM ANALYSIS FILE */
/* JOHN FRASER */

#include <stdio.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/errno.h>
#include <fcntl.h>
#include <mach/machine/vm_param.h>
#include "cache.h"

/* /DEV/MMAP DEFINITIONS */
#define k2phys(addr) (((long)(addr)) & 0xffffffff)
#define SM_MODE (MAP_FILE|MAP_VARIABLE|MAP_SHARED)
#define SM_PROT (PROT_READ|PROT_WRITE)

/* SHARED CACHE DATA POINTER */
datablock* psatom;

/* ADDRESS MAPPING FUNCTIONS */
void FatalError(char* string)
  {
  fprintf(stderr,"ucache: %s\n",string);
  exit(1);
  }

long GetAddress(char* vmunixDebug, char* symbol)
  {
  long addr;
  char command[200];
  int fields;
  FILE* file;
  sprintf(command,"nm -B %s | grep ' %s$",vmunixDebug,symbol);
  file = popen(command,"r");
  if (file==NULL)
    {
    fprintf(stderr,"Open failed: %s\n", command);
```

```
    exit(1);
    }
  fields = fscanf(file,"0x%lx",&addr);
  if (fields!=1) FatalError("Get address failed");
  pclose(file);
  return addr;
  }

/* INITIALIZATION ROUTINE */
void initcache(int proc)
  {
  /* GET POINTER TO SHARED DATA IN KERNEL */
  caddr_t sm_addr;
  size_t length;
  off_t sm_physbase, sm_pgoff;
  unsigned long kbase = GetAddress("vmunix.debug","satom");
  int fd = open("/dev/mem", O_RDWR, 0);
  if (fd<0) FatalError("Unable to open /dev/mem\n");
  sm_physbase = k2phys(alpha_trunc_page(kbase));
  sm_pgoff = kbase & (ALPHA_PGBYTES-1);
  length = alpha_round_page(sm_pgoff + sizeof(datablock));
  sm_addr = mmap(NULL, length, SM_PROT, SM_MODE, fd, sm_physbase);
  if (sm_addr == (caddr_t)-1) FatalError("mmap failed\n");
  psatom = (datablock*) ((long)sm_addr | (long)sm_pgoff);
  /* INCREMENT PROCESS COUNTER */
  psatom->count++;
  /* IF FIRST PROCESS, INITIALIZE CACHE DATA */
  if (proc == 1)
    {
    int tempnumcaches,tempnumtasks;
    int x,a,b,c,d;
    FILE *input, *output;
    /* LOAD BASIC CHARACTERISTICS FROM FILE */
    input = fopen("cache.in","r");
    fgets(psatom->name[0], 79, input);
    fscanf(input,"%d\n",&tempnumtasks);
    for (x=1; x<tempnumtasks; x++)
      fgets(psatom->name[x], 79, input);
    fscanf(input,"%d\n",&tempnumcaches);
    for (x=0; x<tempnumcaches; x++)
      {
      fscanf(input, "%d\n", &(psatom->para[x]).type);
      if ((psatom->para[x]).type == 0)
        fscanf(input, "%d %d %d\n", &(psatom->para[x]).csize[0],
                                    &(psatom->para[x]).lsize[0],
                                    &(psatom->para[x]).assoc[0]);
      else
        fscanf(input,"%d %d %d %d %d %d\n", &(psatom->para[x]).csize[0],
                                            &(psatom->para[x]).lsize[0],
                                            &(psatom->para[x]).assoc[0],
                                            &(psatom->para[x]).csize[1],
```

```
                                        &(psatom->para[x]).lsize[1],
                                        &(psatom->para[x]).assoc[1]);
  }
/* SET ADDRESS HASHING PARAMETERS */
for (a=0; a<tempnumcaches; a++)
  for (b=0; b<((psatom->para[a]).type + 1); b++)
    {
    (psatom->para[a]).tshift[b] = mylog2((psatom->para[a]).csize[b]/
                                         (psatom->para[a]).assoc[b]);
    (psatom->para[a]).lshift[b] = mylog2( (psatom->para[a]).lsize[b] );
    (psatom->para[a]).lmask[b] = ((psatom->para[a]).csize[b]/
                                  (psatom->para[a]).assoc[b])-1;
    }
/* INITIALIZE CACHE STORAGE */
for (a=0; a<tempnumcaches; a++)
  for (b=0; b<((psatom->para[a]).type + 1); b++)
    for (c=0; c<((psatom->para[a]).csize[b]/
                ((psatom->para[a]).lsize[b]*
                 (psatom->para[a]).assoc[b])); c++)
      for (d=0; d<(psatom->para[a]).assoc[b];d++)
        {
        (psatom->data[a][b][c][d]).use = 0;
        (psatom->data[a][b][c][d]).task = tempnumtasks;
        }
/* INITIALIZE CACHE STATISTICS */
for (a=0; a<tempnumcaches; a++)
  for (b=0; b <tempnumtasks; b++)
    {
    (psatom->stat[a][b]).instcnt = 0;
    (psatom->stat[a][b]).readcnt = 0;
    (psatom->stat[a][b]).writcnt = 0;
    (psatom->stat[a][b]).instmisscnt = 0;
    (psatom->stat[a][b]).readmisscnt = 0;
    (psatom->stat[a][b]).writmisscnt = 0;
    for (c=0; c <= tempnumtasks; c++)
      (psatom->stat[a][b]).interfere[c] = 0;
    }
/* LOG SIMULATION DATA TO OUTPUT FILE */
output = fopen("cache.out","w");
fprintf(output,"\n\n\n\n\n\n\n\n");
fprintf(output,"<><><><><><><><><><><><><><><><><><><>\n");
fprintf(output,"SIMULATION: %s",psatom->name[0]);
fprintf(output,"<><><><><><><><><><><><><><><><><><><>\n");
fprintf(output,"\n\n\n\n");
fprintf(output,"Number Tasks = %d\n\n",tempnumtasks);
fprintf(output,"       #0: kernel\n\n");
for (x=1; x<tempnumtasks; x++)
  fprintf(output,"       #%d: %s\n",x,psatom->name[x]);
fprintf(output,"\n\n\n\n");
fprintf(output,"Number Caches = %d\n",tempnumcaches);
fprintf(output,"       (type, icsize, ilsize, iassoc,
```

```
                                dcsize, dlsize, dassoc)\n\n");
    for (x=0; x<tempnumcaches; x++)
      {
      fprintf(output,"       #%d: %1d %7d %5d %3d",x,
                                        (psatom->para[x]).type,
                                        (psatom->para[x]).csize[0],
                                        (psatom->para[x]).lsize[0],
                                        (psatom->para[x]).assoc[0]);
      if ((psatom->para[x]).type == 1)
        fprintf(output," %7d %5d %3d",(psatom->para[x]).csize[1],
                                      (psatom->para[x]).lsize[1],
                                      (psatom->para[x]).assoc[1]);
      fprintf(output,"\n\n");
      }
    fprintf(output,"\f");
    fclose(output);
    /* START CAPTURE & SIMULATION */
    psatom->numtasks = tempnumtasks;
    psatom->numcaches = tempnumcaches;
    psatom->actcaches = tempnumcaches;
    psatom->curtask = -1;
    }
  return;
  }


/* INSTRUCTION REFERENCE ROUTINE */
void instref(long addr, int proc, int count)
  {
  int x, leastx;
  unsigned long leastused;
  long aline, atag;
  int cnum, hit;
  /* PAUSE CAPTURE (RE-ENTRANCE) */
  int tempnumcaches = psatom->numcaches;
  psatom->numcaches = 0;
  /* RE-ESTABLISH AFTER CONTEXT SWTICH (RE-ENTRANCE) */
  if (psatom->curtask != proc)
    {
    tempnumcaches = psatom->actcaches;
    psatom->curtask = proc;
    }
  /* PROCESS REFERENCES IN EACH CACHE */
  for (cnum=0; cnum<tempnumcaches; cnum++)
    {
    int assoc = (psatom->para[cnum]).assoc[0];
    /* UPDATE STATISTICS */
    ((psatom->stat[cnum][proc]).instcnt) += count;
    /* PARSE ADDRESS */
    aline = (addr & (psatom->para[cnum]).lmask[0]) >>
                    (psatom->para[cnum]).lshift[0];
```

```
    atag = addr >> (psatom->para[cnum]).tshift[0];
    /* UPDATE 'USE BITS' AND CHECK FOR HIT */
    hit = 0;
    for (x=0; x<assoc; x++)
      {
      ((psatom->data[cnum][0][aline][x]).use)++;
      if (((psatom->data[cnum][0][aline][x]).tag == atag) &&
          ((psatom->data[cnum][0][aline][x]).task == proc))
        {
        (psatom->data[cnum][0][aline][x]).use = 0;
        hit = 1;
        }
      }
    /* IF NOT HIT, FIND LRU BLOCK TO EVICT */
    if (hit == 0)
      {
      /* FIND LRU */
      leastused = 0;
      for (x=0; x<assoc; x++)
        {
        if (((psatom->data[cnum][0][aline][x]).use >= leastused) ||
            ((psatom->data[cnum][0][aline][x]).task ==
                                                psatom->numtasks))
          {
          leastused = (psatom->data[cnum][0][aline][x]).use;
          leastx = x;
          }
        if ((psatom->data[cnum][0][aline][x]).task ==
                                                psatom->numtasks)
          x = assoc;
        }
      /* UPDATE STATISTICS */
      ((psatom->stat[cnum][proc]).instmisscnt)++;
      ((psatom->stat[cnum][proc]).interfere[
       (psatom->data[cnum][0][aline][leastx]).task])++;
      /* UPDATE CACHE DATA */
      (psatom->data[cnum][0][aline][leastx]).tag = atag;
      (psatom->data[cnum][0][aline][leastx]).use = 0;
      (psatom->data[cnum][0][aline][leastx]).task = proc;
      }
    }
  /* RESUME CAPTURE */
  psatom->numcaches = tempnumcaches;
  return;
  }

/* DATA LOAD ROUTINE */
void readref(long addr, int proc)
  {
  int index;
  int x, leastx;
```

```
unsigned long leastused;
long aline, atag;
int cnum, hit;
/* PAUSE CAPTURE (RE-ENTRANCE) */
int tempnumcaches = psatom->numcaches;
psatom->numcaches = 0;
/* RE-ESTABLISH AFTER CONTEXT SWITCH (RE-ENTRANCE) */
if (psatom->curtask != proc)
  {
  tempnumcaches = psatom->actcaches;
  psatom->curtask = proc;
  }
/* PROCESS REFERENCE IN EACH CACHE */
for (cnum=0; cnum<tempnumcaches; cnum++)
  {
  int type = (psatom->para[cnum]).type;
  int assoc = (psatom->para[cnum]).assoc[type];
  /* UPDATE STATISTICS */
  ((psatom->stat[cnum][proc]).readcnt)++;
  /* PARSE ADDRESS */
  aline = (addr & (psatom->para[cnum]).lmask[type]) >>
                  (psatom->para[cnum]).lshift[type];
  atag = addr >> (psatom->para[cnum]).tshift[type];
  /* UPDATE 'USE BITS' AND CHECK FOR HIT */
  hit = 0;
  for (x=0; x<assoc; x++)
    {
    ((psatom->data[cnum][type][aline][x]).use)++;
    if (((psatom->data[cnum][type][aline][x]).tag == atag) &&
        ((psatom->data[cnum][type][aline][x]).task == proc))
      {
      (psatom->data[cnum][type][aline][x]).use = 0;
      hit = 1;
      }
    }
  /* IF NO HIT, FIND LRU BLOCK TO EVICT */
  if (hit == 0)
    {
    /* FIND LRU */
    leastused = 0;
    for (x=0; x<assoc; x++)
      {
      if (((psatom->data[cnum][type][aline][x]).use >= leastused) ||
          ((psatom->data[cnum][type][aline][x]).task ==
                                                psatom->numtasks))
        {
        leastused = (psatom->data[cnum][type][aline][x]).use;
        leastx = x;
        }
      if ((psatom->data[cnum][type][aline][x]).task ==
                                                psatom->numtasks)
```

```
          x = assoc;
        }
      /* UPDATE STATISTICS */
      ((psatom->stat[cnum][proc]).readmisscnt)++;
      ((psatom->stat[cnum][proc]).interfere[
       (psatom->data[cnum][type][aline][leastx]).task])++;
      /* UPDATE CACHE DATA */
      (psatom->data[cnum][type][aline][leastx]).tag = atag;
      (psatom->data[cnum][type][aline][leastx]).use = 0;
      (psatom->data[cnum][type][aline][leastx]).task = proc;
      }
    }
  /* RESUME CAPTURE */
  psatom->numcaches = tempnumcaches;
  return;
  }

/* DATA STORE ROUTINE */
void writref(long addr, int proc)
  {
  int index;
  int x, leastx;
  unsigned long leastused;
  long aline, atag;
  int cnum, hit;
  /* PAUSE CAPTURE (RE-ENTRANCE) */
  int tempnumcaches = psatom->numcaches;
  psatom->numcaches = 0;
  /* RE-ESTABLISH AFTER CONTEXT SWTICH (RE-ENTRANCE) */
  if (psatom->curtask != proc)
    {
    tempnumcaches = psatom->actcaches;
    psatom->curtask = proc;
    }
  /* PROCESS REFERENCE IN EACH CACHE */
  for (cnum=0; cnum<tempnumcaches; cnum++)
    {
    int type = (psatom->para[cnum]).type;
    int assoc = (psatom->para[cnum]).assoc[type];
    /* UPDATE STATISTICS */
    ((psatom->stat[cnum][proc]).writcnt)++;
    /* PARSE ADDRESS */
    aline = (addr & (psatom->para[cnum]).lmask[type]) >>
                    (psatom->para[cnum]).lshift[type];
    atag = addr >> (psatom->para[cnum]).tshift[type];
    /* UPDATE 'USE BITS' AND CHECK FOR HIT */
    hit = 0;
    for (x=0; x<assoc; x++)
      {
      ((psatom->data[cnum][type][aline][x]).use)++;
      if (((psatom->data[cnum][type][aline][x]).tag == atag) &&
```

```
          ((psatom->data[cnum][type][aline][x]).task == proc))
        {
        (psatom->data[cnum][type][aline][x]).use = 0;
        hit = 1;
        }
      }
    /* IF NOT HIT, FIND LRU BLOCK TO EVICT */
    if (hit == 0)
      {
      /* FIND LRU */
      leastused = 0;
      for (x=0; x<assoc; x++)
        {
        if (((psatom->data[cnum][type][aline][x]).use >= leastused) ||
            ((psatom->data[cnum][type][aline][x]).task ==
                                                  psatom->numtasks))
          {
          leastused = (psatom->data[cnum][type][aline][x]).use;
          leastx = x;
          }
        if ((psatom->data[cnum][type][aline][x]).task ==
                                                  psatom->numtasks)
          x = assoc;
        }
      /* UPDATE STATISTICS */
      ((psatom->stat[cnum][proc]).writmisscnt)++;
      ((psatom->stat[cnum][proc]).interfere[
       (psatom->data[cnum][type][aline][leastx]).task])++;
      /* UPDATE CACHE DATA */
      (psatom->data[cnum][type][aline][leastx]).tag = atag;
      (psatom->data[cnum][type][aline][leastx]).use = 0;
      (psatom->data[cnum][type][aline][leastx]).task = proc;
      }
    }
  /* RESUME CAPTURE */
  psatom->numcaches = tempnumcaches;
  return;
  }

/* STORE RESULTS ROUTINE */
void printres(int proc)
  {
  int c,x,y;
  stats total;
  FILE* file;
  /* PAUSE CAPTURE */
  int tempnumcaches = psatom->actcaches;
  psatom->numcaches = 0;
  /* OPEN FILE FOR OUTPUT */
  file = fopen("cache.out","a");
  fprintf(file,"DATA AT END OF PROCESS %d\n",proc);
```

```
fprintf(file,"<><><><><><><><><><><><><><><><><><><><><><><><><><><><><><>\n");
/* PRINT DATA FOR EACH CACHE */
for (c=0; c<tempnumcaches; c++)
  {
  fprintf(file,"simulation: %s            (data at end of process %d)\n",
                                                  psatom->name[0],proc);
  fprintf(file,"----------------------------------------------\n");
  fprintf(file,"CACHE # %d\n", c);
  fprintf(file,"cache type: %d (0=unified, 1=split)\n",
                                          (psatom->para[c]).type);
  fprintf(file,"icache size: %d\n",(psatom->para[c]).csize[0]);
  fprintf(file,"icache line size: %d\n",(psatom->para[c]).lsize[0]);
  fprintf(file,"icache associativity: %d\n",
                                        (psatom->para[c]).assoc[0]);
  if ((psatom->para[c]).type == 1)
    {
    fprintf(file,"dcache size: %d\n",(psatom->para[c]).csize[1]);
    fprintf(file,"dcache line size: %d\n",(psatom->para[c]).lsize[1]);
    fprintf(file,"dcache associativity: %d\n",
                                        (psatom->para[c]).assoc[1]);
    }
  total.instcnt = 0;
  total.readcnt = 0;
  total.writcnt = 0;
  total.instmisscnt = 0;
  total.readmisscnt = 0;
  total.writmisscnt = 0;
  /* PRINT PROCESS CACHE PERFORMANCE *./
  for (y=0; y < psatom->numtasks; y++)
    {
    int z;
    total.instcnt = total.instcnt + (psatom->stat[c][y]).instcnt;
    total.readcnt = total.readcnt + (psatom->stat[c][y]).readcnt;
    total.writcnt = total.writcnt + (psatom->stat[c][y]).writcnt;
    total.instmisscnt = total.instmisscnt +
                        (psatom->stat[c][y]).instmisscnt;
    total.readmisscnt = total.readmisscnt +
                        (psatom->stat[c][y]).readmisscnt;
    total.writmisscnt = total.writmisscnt +
                        (psatom->stat[c][y]).writmisscnt;
    fprintf(file,"     **********\n");
    fprintf(file,"     Process #%d\n", y);
    fprintf(file,"         Inst  %12lu ", (psatom->stat[c][y]).instcnt);
    fprintf(file,"Miss %12lu ", (psatom->stat[c][y]).instmisscnt);
    if ((psatom->stat[c][y]).instcnt != 0)
      fprintf(file,"Perc %.6lf", 100.0 *
                              (psatom->stat[c][y]).instmisscnt /
                              (psatom->stat[c][y]).instcnt);
    fprintf(file,"\n        Data  %12lu ", (psatom->stat[c][y]).readcnt +
                                        (psatom->stat[c][y]).writcnt);
    fprintf(file,"Miss %12lu ", (psatom->stat[c][y]).readmisscnt +
```

```
                              (psatom->stat[c][y]).writmisscnt);
    if (((psatom->stat[c][y]).readcnt+(psatom->stat[c][y]).writcnt) != 0)
      fprintf(file,"Perc %.6lf", 100.0 *
                              ((psatom->stat[c][y]).readmisscnt +
                               (psatom->stat[c][y]).writmisscnt) /
                              ((psatom->stat[c][y]).readcnt +
                               (psatom->stat[c][y]).writcnt));
    fprintf(file,"\n          read %12lu ",
                              (psatom->stat[c][y]).readcnt);
    fprintf(file,"Miss %12lu ", (psatom->stat[c][y]).readmisscnt);
    if ((psatom->stat[c][y]).readcnt != 0)
      fprintf(file,"Perc %.6lf", 100.0 *
                              (psatom->stat[c][y]).readmisscnt /
                              (psatom->stat[c][y]).readcnt);
    fprintf(file,"\n          writ %12lu ", (psatom->stat[c][y]).writcnt);
    fprintf(file,"Miss %12lu ", (psatom->stat[c][y]).writmisscnt);
    if ((psatom->stat[c][y]).writcnt != 0)
      fprintf(file,"Perc %.6lf", 100.0 *
                              (psatom->stat[c][y]).writmisscnt /
                              (psatom->stat[c][y]).writcnt);
    fprintf(file,"\n         TOTAL %12lu ", (psatom->stat[c][y]).instcnt +
                                             (psatom->stat[c][y]).readcnt +
                                             (psatom->stat[c][y]).writcnt);
    fprintf(file,"Miss %12lu ", (psatom->stat[c][y]).instmisscnt +
                                (psatom->stat[c][y]).readmisscnt +
                                (psatom->stat[c][y]).writmisscnt);
    if (((psatom->stat[c][y]).instcnt +
         (psatom->stat[c][y]).readcnt +
         (psatom->stat[c][y]).writcnt) != 0)
      fprintf(file,"Perc %.6lf", 100.0 *
                              ((psatom->stat[c][y]).instmisscnt +
                               (psatom->stat[c][y]).readmisscnt +
                               (psatom->stat[c][y]).writmisscnt) /
                              ((psatom->stat[c][y]).instcnt +
                               (psatom->stat[c][y]).readcnt +
                               (psatom->stat[c][y]).writcnt));
    fprintf(file,"\n          Int (times process %d overwrote:)\n", y);
    for (z=0; z <= psatom->numtasks; z++)
      fprintf(file,"               Process %d = %12lu\n", z,
                              (psatom->stat[c][y]).interfere[z]);
    fprintf(file,"             (process %d is invalid data)\n",
                                             psatom->numtasks);
    }
  /* PRINT TOTAL CACHE PERFORMANCE */
  fprintf(file,"     ********************\n");
  fprintf(file,"     TOTAL FOR CACHE\n");
  fprintf(file,"         Inst  %12lu ", total.instcnt);
  fprintf(file,"Miss %12lu ", total.instmisscnt);
  if (total.instcnt != 0)
    fprintf(file,"Perc %.6lf", 100.0 * total.instmisscnt /
                                  total.instcnt);
```

```
  fprintf(file,"\n         Data  %12lu ", total.readcnt +
                                          total.writcnt);
  fprintf(file,"Miss %12lu ", total.readmisscnt + total.writmisscnt);
  if ((total.readcnt + total.writcnt) != 0)
    fprintf(file,"Perc %.6lf", 100.0 *
                               (total.readmisscnt + total.writmisscnt)/
                               (total.readcnt + total.writcnt));
  fprintf(file,"\n          read %12lu ", total.readcnt);
  fprintf(file,"Miss %12lu ", total.readmisscnt);
  if (total.readcnt != 0)
    fprintf(file,"Perc %.6lf", 100.0 * total.readmisscnt /
                                       total.readcnt);
  fprintf(file,"\n          writ %12lu ", total.writcnt);
  fprintf(file,"Miss %12lu ", total.writmisscnt);
  if (total.writcnt != 0)
    fprintf(file,"Perc %.6lf", 100.0 * total.writmisscnt /
                                       total.writcnt);
  fprintf(file,"\n         TOTAL %12lu ", total.instcnt +
                                          total.readcnt +
                                          total.writcnt);
  fprintf(file,"Miss %12lu ", total.instmisscnt +
                              total.readmisscnt +
                              total.writmisscnt);
  if ((total.instcnt + total.readcnt + total.writcnt) != 0)
    fprintf(file,"Perc %.6lf", 100.0 *
                               (total.instmisscnt +
                                total.readmisscnt +
                                total.writmisscnt) /
                               (total.instcnt +
                                total.readcnt +
                                total.writcnt));
  fprintf(file,"\n");
  fprintf(file,"\f");
  }
fclose(file);
/* IF LAST PROCESS, SHUT DOWN SIMULATION */
psatom->count--;
if (psatom->count > 0)
  {
  psatom->numcaches = tempnumcaches;
  psatom->curtask = proc;
  }
return;
}
```

A.8 Sample Tool Description File

To create an ATOM tool, a tool description file must be created which defines the various tool characteristics such as the files to incorporate and control flags to use. An example is shown below, which is the tool used to create the executable version of the kernel `kexe.desc`. For more information, please refer to the ATOM source documents.

```
INST_FILE       kern.inst.c

ANAL_FILE       kern.anal.c

ANAL_LDFLAGS    -non_shared

ATOM_REQ        -Xkernel -Xgprog

ATOM_DEF        -o vmunix.cache
```

Another tool example is the one used for the context switch model, `mod.desc`, which shows the `-lm` flag required to use functions from the `libm.a` library.

```
INST_FILE       prog.inst.c

ANAL_FILE       model.anal.c

ANAL_LDFLAGS    -lm
```

A.9 Model Library

The following file, model.h, was used as a procedure library for the context switch model implementation. It is used in conjunction with the cache model library.

```
/* MODEL.H */
/* CONTEXT SWTICH MODEL LIBRARY */
/* JOHN FRASER */

#include <stdlib.h>
#include <math.h>

/* COMPUTE RANDOM EXECUTION INTERVAL */
long compint()
  {
  long temp = random();
  temp = (long)trunc(-50000.0*log(1.0-(random()/(pow(2.0,31.0)-1.0))));
  /* INTERVAL CAP */
  if (temp > 250000)
    return(250000);
  else
    return(temp);
  }

/* COMPUTE FACTORIAL FUNCTION */
double myfact(long x)
  {
  if (x == 0)
    return(1.0);
  else
    return((double) x * myfact(x-1));
  }

/* COMPUTE COMBINATORIAL FUNCTION */
double mycomb(long F, long i)
  {
  long x;
  double temp3 = 1.0/myfact(i);
  /* CANT USE STANDARD FACTORIAL EXPRESSION => OVERFLOW ERROR */
  for (x=F; x>F-i; x--)
    temp3 = temp3 * x;
  return(temp3);
  }

/* COMPUTE BLOCK OVERWRITE PROBABILITY */
double calcprob(long F, int C, int B, int A, int i)
  {
  int x;
  double temp2 = 0.0;
  int N = C/(B*A);
  if (i < A)
    {
```

```
    double a,b,c;
    a = (double)(mycomb(F,i));
    b = (double)(pow((1.0/(double)N),(double)i));
    /* UNDERFLOW TEST FOR LAST TERM */
    if ((F-i)*log(1.0-(1.0/(double)N)) < -600.0)
      c = 0;
    else
      c = (double)pow((1.0-(1.0/(double)N)),((double)(F-i)));
    return(a*b*c);
    }
  else
    for (x=0; x < A; x++)
      temp2 = temp2 + ((double)(mycomb(F,x)) *
                       (pow((1.0/N),x)) *
                       (pow((1.0-(1.0/N)),(F-x))));
    return(1.0 - temp2);
  }

/* COMPUTE INSTRUCTION FOOTPRINT */
long ifoot(long R, int B)
  {
  return((long)trunc(R/(50.0*B)));
  }

/* COMPUTE DATA FOOTPRINT */
long dfoot(long R)
  {
  return((long)trunc(R/50.0));
  }
```

A.10 Model Analysis File

The files used to test the context switch model were very similar to those used in the first set of simulations. The program instrumentation file was identical, and the analysis file `model.anal.c` was generally the same, although with the addition of the model code as shown. Since the model was tested with a single process trace, the re-entrance mechanisms were not required.

```
/* MODEL.ANAL.C */
/* PROGRAM ANALYSIS FILE */
/* W/ CONTEXT SWITCH MODEL */
/* JOHN FRASER */

#include <stdio.h>
#include "cache.h"
#include "model.h"

/* CACHE DATA */
datablock satom;
datablock* psatom;

/* MODEL DATA */
unsigned long switchnext;
unsigned long switchcnt;
unsigned long switchrec;

/* INITIALIZATION ROUTINE */
void initcache(int proc)
  {
  /* SET POINTER TO CACHE DATA */
  psatom = &satom;
  /* INITIALIZE BASIC DATA */
  psatom->count = 0;
  psatom->numcaches = 0;
  psatom->numtasks = 0;
  /* INITIALIZE SWITCH MODEL */
  switchcnt = 0;
  switchrec = 0;
  switchnext = compint();
  /* IF FIRST PROCESS, INITIALIZE CACHE DATA */
  psatom->count++;
  if (psatom->count == 1)
    {
    int tempnumcaches,tempnumtasks;
    int x,a,b,c,d;
    FILE *input, *output;
    /* LOAD BASIC CHARACTERISTICS FROM FILE */
    input = fopen("cache.in","r");
    fgets(psatom->name[0], 79, input);
    fscanf(input,"%d\n",&tempnumtasks);
    for (x=1; x<tempnumtasks; x++)
      fgets(psatom->name[x], 79, input);
    fscanf(input,"%d\n",&tempnumcaches);
```

```
for (x=0; x<tempnumcaches; x++)
  {
  fscanf(input, "%d\n", &(psatom->para[x]).type);
  if ((psatom->para[x]).type == 0)
    fscanf(input, "%d %d %d\n", &(psatom->para[x]).csize[0],
                               &(psatom->para[x]).bsize[0],
                               &(psatom->para[x]).assoc[0]);
  else
    fscanf(input,"%d %d %d %d %d %d\n", &(psatom->para[x]).csize[0],
                                        &(psatom->para[x]).bsize[0],
                                        &(psatom->para[x]).assoc[0],
                                        &(psatom->para[x]).csize[1],
                                        &(psatom->para[x]).bsize[1],
                                        &(psatom->para[x]).assoc[1]);
  }
/* SET ADDRESS HASHING PARAMETERS */
for (a=0; a<tempnumcaches; a++)
  for (b=0; b<((psatom->para[a]).type + 1); b++)
    {
    (psatom->para[a]).tshift[b] = mylog2((psatom->para[a]).csize[b]/
                                         (psatom->para[a]).assoc[b]);
    (psatom->para[a]).lshift[b] = mylog2((psatom->para[a]).bsize[b]);
    (psatom->para[a]).lmask[b] = ((psatom->para[a]).csize[b]/
                                  (psatom->para[a]).assoc[b])-1;
    }
/* INITIALIZE CACHE STORAGE */
for (a=0; a<tempnumcaches; a++)
  for (b=0; b<((psatom->para[a]).type + 1); b++)
    for (c=0; c<((psatom->para[a]).csize[b] /
                ((psatom->para[a]).bsize[b] *
                 (psatom->para[a]).assoc[b])); c++)
      for (d=0; d<(psatom->para[a]).assoc[b];d++)
        {
        (psatom->data[a][b][c][d]).use = 0;
        (psatom->data[a][b][c][d]).task = tempnumtasks;
        }
/* INITIALIZE CACHE STATISTICS */
for (a=0; a<tempnumcaches; a++)
  for (b=0; b <tempnumtasks; b++)
    {
    (psatom->stat[a][b]).instcnt = 0;
    (psatom->stat[a][b]).readcnt = 0;
    (psatom->stat[a][b]).writcnt = 0;
    (psatom->stat[a][b]).instmisscnt = 0;
    (psatom->stat[a][b]).readmisscnt = 0;
    (psatom->stat[a][b]).writmisscnt = 0;
    for (c=0; c <= tempnumtasks; c++)
      (psatom->stat[a][b]).interfere[c] = 0;
    }
/* LOG SIMULATION DATA TO OUTPUT FILE */
output = fopen("cache.out","w");
```

```
    fprintf(output,"\n\n\n\n\n\n\n\n\n");
    fprintf(output,"<><><><><><><><><><><><><><><><><><><><><><><><><><><><>\n");
    fprintf(output,"SIMULATION (single): %s",psatom->name[0]);
    fprintf(output,"<><><><><><><><><><><><><><><><><><><><><><><><><><><><>\n");
    fprintf(output,"\n\n\n\n");
    fprintf(output,"Number Tasks = %d\n\n",tempnumtasks);
    for (x=1; x<tempnumtasks; x++)
      fprintf(output,"       #%d: %s\n",x,psatom->name[x]);
    fprintf(output,"\n\n\n\n");
    fprintf(output,"Number Caches = %d\n",tempnumcaches);
    fprintf(output,"       (type, icsize, ibsize, iassoc,
                              dcsize, dbsize, dassoc)\n\n");
    for (x=0; x<tempnumcaches; x++)
      {
      fprintf(output,"       #%d: %1d %7d %5d %3d",x,
                                          (psatom->para[x]).type,
                                          (psatom->para[x]).csize[0],
                                          (psatom->para[x]).bsize[0],
                                          (psatom->para[x]).assoc[0]);
      if ((psatom->para[x]).type == 1)
        fprintf(output," %7d %5d %3d",(psatom->para[x]).csize[1],
                                      (psatom->para[x]).bsize[1],
                                      (psatom->para[x]).assoc[1]);
      fprintf(output,"\n\n");
      }
    fprintf(output,"\f");
    fclose(output);
    /* START SIMULATION */
    psatom->numtasks = tempnumtasks;
    psatom->numcaches = tempnumcaches;
    }
  return;
  }


/* INSTRUCTION REFERENCE ROUTINE */
void instref(long addr, int proc, int count)
  {
  int x, leastx;
  unsigned long leastused;
  long aline, atag;
  int cnum, hit;
  /* PROCESS REFERENCES IN EACH CACHE */
  for (cnum=0; cnum < psatom->numcaches; cnum++)
    {
    int assoc = (psatom->para[cnum]).assoc[0];
    /* UPDATE STATISTICS */
    ((psatom->stat[cnum][proc]).instcnt) += count;
    /* PARSE ADDRESS */
    aline = (addr & (psatom->para[cnum]).lmask[0]) >>
                    (psatom->para[cnum]).lshift[0];
```

```
    atag = addr >> (psatom->para[cnum]).tshift[0];
    /* UPDATE 'USE BITS' AND CHECK FOR HIT */
    hit = 0;
    for (x=0; x<assoc; x++)
      {
      ((psatom->data[cnum][0][aline][x]).use)++;
      if (((psatom->data[cnum][0][aline][x]).tag == atag) &&
          ((psatom->data[cnum][0][aline][x]).task == proc))
        {
        (psatom->data[cnum][0][aline][x]).use = 0;
        hit = 1;
        }
      }
    /* IF NO HIT, FIND LRU BLOCK TO EVICT */
    if (hit == 0)
      {
      /* FIND LRU */
      leastused = 0;
      for (x=0; x<assoc; x++)
        {
        if (((psatom->data[cnum][0][aline][x]).use >= leastused) ||
            ((psatom->data[cnum][0][aline][x]).task ==
                                                psatom->numtasks))
          {
          leastused = (psatom->data[cnum][0][aline][x]).use;
          leastx = x;
          }
        if ((psatom->data[cnum][0][aline][x]).task ==
                                                psatom->numtasks)
          x = assoc;
        }
      /* UPDATE STATISTICS */
      ((psatom->stat[cnum][proc]).instmisscnt)++;
      ((psatom->stat[cnum][proc]).interfere[
       (psatom->data[cnum][0][aline][leastx]).task])++;
      /* UPDATE CACHE DATA */
      (psatom->data[cnum][0][aline][leastx]).tag = atag;
      (psatom->data[cnum][0][aline][leastx]).use = 0;
      (psatom->data[cnum][0][aline][leastx]).task = proc;
      }
    }
  /* INCREMENT SWTICH COUNTER */
  switchcnt += count;
  /* CHECK FOR CONTEXT SWTICH AND PERFORM */
  if (switchcnt >= switchnext)
    {
    unsigned long intercnt;
    long foot;
    int sec;
    double prob,prbcnt;
    /* COMPUTE INTERRUPTION INTERVAL */
```

```
intercnt = (psatom->numtasks-1) * compint();
/* APPLY IMPACT TO EACH CACHE */
for (cnum=0; cnum < psatom->numcaches; cnum++)
  {
  /* APPLY IMPACT TO EACH SECTION (INST/DATA) */
  for (sec=0; sec<=(psatom->para[cnum]).type; sec++)
    {
    /* COMPUTE FOOTPRINT FOR EACH SECTION */
    if (sec==0)
      {
      foot = ifoot(intercnt, ((psatom->para[cnum]).bsize[sec] / 4));
      if ((psatom->para[cnum]).type == 0)
        foot = foot + dfoot(intercnt);
      }
    else
      foot = dfoot(intercnt);
    /* ITERATE THROUGH EACH LINE OVERWRITING RANDOM BLOCK(S) */
    for (aline=0; aline < (psatom->para[cnum]).csize[sec] /
                          ((psatom->para[cnum]).bsize[sec] *
                           (psatom->para[cnum]).assoc[sec]); aline++)
      {
      /* GENERATE LINE'S PROBABILITY */
      prob = (double)random()/(pow(2.0,31.0)-1.0);
      /* COMPUTE PROBABILITY OF FIRST OVERWRITE */
      prbcnt = calcprob(foot,
                        (psatom->para[cnum]).csize[sec],
                        (psatom->para[cnum]).bsize[sec],
                        (psatom->para[cnum]).assoc[sec],
                        0);
      /* ITERATE UNTIL ALL OVERWRITTEN OR PROBABILITY FAILS */
      for (hit=0; ((hit < (psatom->para[cnum]).assoc[sec]) &&
                   (prob > prbcnt)); hit++)
        {
        /* COMPUTE PROBABILITY OF NEXT OVERWRITE */
        if (hit < ((psatom->para[cnum]).assoc[sec] - 1))
          prbcnt += calcprob(foot,
                             (psatom->para[cnum]).csize[sec],
                             (psatom->para[cnum]).bsize[sec],
                             (psatom->para[cnum]).assoc[sec],
                             hit+1);
        /* FIND LRU BLOCK TO EVICT */
        leastused = 0;
        for (x=0; x < (psatom->para[cnum]).assoc[sec]; x++)
          {
          /* UPDATE 'USE BITS' */
          (psatom->data[cnum][sec][aline][x]).use++;
          if ((psatom->data[cnum][sec][aline][x]).use >= leastused)
            {
            leastused = (psatom->data[cnum][sec][aline][x]).use;
            leastx = x;
            }
```

```
            }
          /* UPDATE CACHE DATA */
          (psatom->data[cnum][sec][aline][leastx]).use = 0;
          (psatom->data[cnum][sec][aline][leastx]).task =
                      (psatom->numtasks - 1);
          }
        }
      }
    }
  /* RESET FOR NEXT INTERVAL */
  switchrec++;
  switchcnt = 0;
  switchnext = compint();
  }
return;
}

/* DATA LOAD ROUTINE */
void readref(long addr, int proc)
  {
  int index;
  int x, leastx;
  unsigned long leastused;
  long aline, atag;
  int cnum, hit;
  /* PROCESS REFERENCE IN EACH CACHE */
  for (cnum=0; cnum<psatom->numcaches; cnum++)
    {
    int type = (psatom->para[cnum]).type;
    int assoc = (psatom->para[cnum]).assoc[type];
    /* UPDATE STATISTICS */
    ((psatom->stat[cnum][proc]).readcnt)++;
    /* PARSE ADDRESS */
    aline = (addr & (psatom->para[cnum]).lmask[type]) >>
                    (psatom->para[cnum]).lshift[type];
    atag = addr >> (psatom->para[cnum]).tshift[type];
    /* UPDATE 'USE BITS' AND CHECK FOR HIT */
    hit = 0;
    for (x=0; x<assoc; x++)
      {
      ((psatom->data[cnum][type][aline][x]).use)++;
      if (((psatom->data[cnum][type][aline][x]).tag == atag) &&
          ((psatom->data[cnum][type][aline][x]).task == proc))
        {
        (psatom->data[cnum][type][aline][x]).use = 0;
        hit = 1;
        }
      }
    /* IF NO HIT, FIND LRU BLOCK TO EVICT */
    if (hit == 0)
      {
```

```
      /* FIND LRU */
      leastused = 0;
      for (x=0; x<assoc; x++)
        {
        if (((psatom->data[cnum][type][aline][x]).use >= leastused) ||
            ((psatom->data[cnum][type][aline][x]).task ==
                                                    psatom->numtasks))
          {
          leastused = (psatom->data[cnum][type][aline][x]).use;
          leastx = x;
          }
        if ((psatom->data[cnum][type][aline][x]).task ==
                                                   psatom->numtasks)
          x = assoc;
        }
      /* UPDATE STATISTICS */
      ((psatom->stat[cnum][proc]).readmisscnt)++;
      ((psatom->stat[cnum][proc]).interfere[
       (psatom->data[cnum][type][aline][leastx]).task])++;
      /* UPDATE CACHE DATA */
      (psatom->data[cnum][type][aline][leastx]).tag = atag;
      (psatom->data[cnum][type][aline][leastx]).use = 0;
      (psatom->data[cnum][type][aline][leastx]).task = proc;
      }
    }
  return;
  }

/* DATA STORE ROUTINE */
void writref(long addr, int proc)
  {
  int index;
  int x, leastx;
  unsigned long leastused;
  long aline, atag;
  int cnum, hit;
  /* PROCESS REFERENCE IN EACH CACHE */
  for (cnum=0; cnum<psatom->numcaches; cnum++)
    {
    int type = (psatom->para[cnum]).type;
    int assoc = (psatom->para[cnum]).assoc[type];
    /* UPDATE STATISTICS */
    ((psatom->stat[cnum][proc]).writcnt)++;
    /* PARSE ADDRESS */
    aline = (addr & (psatom->para[cnum]).lmask[type]) >>
                    (psatom->para[cnum]).lshift[type];
    atag = addr >> (psatom->para[cnum]).tshift[type];
    /* UPDATE 'USE BITS' AND CHECK FOR HIT */
    hit = 0;
    for (x=0; x<assoc; x++)
      {
```

```
      ((psatom->data[cnum][type][aline][x]).use)++;
      if (((psatom->data[cnum][type][aline][x]).tag == atag) &&
          ((psatom->data[cnum][type][aline][x]).task == proc))
        {
        (psatom->data[cnum][type][aline][x]).use = 0;
        hit = 1;
        }
      }
    /* IF NO HIT, FIND LRU BLOCK TO EVICT */
    if (hit == 0)
      {
      /* FIND LRU BLOCK */
      leastused = 0;
      for (x=0; x<assoc; x++)
        {
        if (((psatom->data[cnum][type][aline][x]).use >= leastused) ||
            ((psatom->data[cnum][type][aline][x]).task ==
                                                psatom->numtasks))
          {
          leastused = (psatom->data[cnum][type][aline][x]).use;
          leastx = x;
          }
        if ((psatom->data[cnum][type][aline][x]).task ==
                                                psatom->numtasks)
          x = assoc;
        }
      /* UPDATE STATISTICS */
      ((psatom->stat[cnum][proc]).writmisscnt)++;
      ((psatom->stat[cnum][proc]).interfere[
       (psatom->data[cnum][type][aline][leastx]).task])++;
      /* UPDATE */
      (psatom->data[cnum][type][aline][leastx]).tag = atag;
      (psatom->data[cnum][type][aline][leastx]).use = 0;
      (psatom->data[cnum][type][aline][leastx]).task = proc;
      }
    }
  return;
  }

/* STORE RESULTS ROUTINE */
void printres(int proc)
  {
  int c,x,y;
  stats total;
  FILE* file;
  file = fopen("cache.out","a");
  fprintf(file,"DATA AT END OF PROCESS %d\n",proc);
  fprintf(file,"<><><><><><><><><><><><><><><><><><><><><><><><><><><><><>\n");
  for (c=0; c<psatom->numcaches; c++)
    {
    /* PRINT CACHE DATA */
```

```
fprintf(file,"simulation: %s         (data at end of process %d)\n",
                                                psatom->name[0],proc);
fprintf(file,"total context switches modeled: %lu\n",switchrec);
fprintf(file,"-----------------------------------------------\n");
fprintf(file,"CACHE # %d\n", c);
fprintf(file,"cache type: %d (0=unified, 1=split)\n",
                                          (psatom->para[c]).type);
fprintf(file,"icache size: %d\n",(psatom->para[c]).csize[0]);
fprintf(file,"icache line size: %d\n",(psatom->para[c]).bsize[0]);
fprintf(file,"icache associativity: %d\n",
                                       (psatom->para[c]).assoc[0]);
if ((psatom->para[c]).type == 1)
  {
  fprintf(file,"dcache size: %d\n",(psatom->para[c]).csize[1]);
  fprintf(file,"dcache line size: %d\n",(psatom->para[c]).bsize[1]);
  fprintf(file,"dcache associativity: %d\n",
                                         (psatom->para[c]).assoc[1]);
  }
total.instcnt = 0;
total.readcnt = 0;
total.writcnt = 0;
total.instmisscnt = 0;
total.readmisscnt = 0;
total.writmisscnt = 0;
/* PRINT PROCESS CACHE PERFORMANCE */
for (y=0; y < psatom->numtasks; y++)
  {
  int z;
  total.instcnt = total.instcnt + (psatom->stat[c][y]).instcnt;
  total.readcnt = total.readcnt + (psatom->stat[c][y]).readcnt;
  total.writcnt = total.writcnt + (psatom->stat[c][y]).writcnt;
  total.instmisscnt = total.instmisscnt +
                      (psatom->stat[c][y]).instmisscnt;
  total.readmisscnt = total.readmisscnt +
                      (psatom->stat[c][y]).readmisscnt;
  total.writmisscnt = total.writmisscnt +
                      (psatom->stat[c][y]).writmisscnt;
  fprintf(file,"     **********\n");
  fprintf(file,"     Process #%d\n", y);
  fprintf(file,"         Inst  %12lu ",(psatom->stat[c][y]).instcnt);
  fprintf(file,"Miss %12lu ", (psatom->stat[c][y]).instmisscnt);
  if ((psatom->stat[c][y]).instcnt != 0)
    fprintf(file,"Perc %.6lf", 100.0*
                               (psatom->stat[c][y]).instmisscnt /
                               (psatom->stat[c][y]).instcnt);
  fprintf(file,"\n         Data  %12lu ",
                               (psatom->stat[c][y]).readcnt +
                               (psatom->stat[c][y]).writcnt);
  fprintf(file,"Miss %12lu ", (psatom->stat[c][y]).readmisscnt +
                              (psatom->stat[c][y]).writmisscnt);
  if (((psatom->stat[c][y]).readcnt +
```

```
        (psatom->stat[c][y]).writcnt) != 0)
      fprintf(file,"Perc %.61f", 100.0 *
                                 ((psatom->stat[c][y]).readmisscnt +
                                  (psatom->stat[c][y]).writmisscnt) /
                                 ((psatom->stat[c][y]).readcnt +
                                  (psatom->stat[c][y]).writcnt));
    fprintf(file,"\n          read %12lu ",
                                 (psatom->stat[c][y]).readcnt);
    fprintf(file,"Miss %12lu ", (psatom->stat[c][y]).readmisscnt);
    if ((psatom->stat[c][y]).readcnt != 0)
      fprintf(file,"Perc %.61f", 100.0 *
                                 (psatom->stat[c][y]).readmisscnt /
                                 (psatom->stat[c][y]).readcnt);
    fprintf(file,"\n          writ %12lu ",
                                 (psatom->stat[c][y]).writcnt);
    fprintf(file,"Miss %12lu ", (psatom->stat[c][y]).writmisscnt);
    if ((psatom->stat[c][y]).writcnt != 0)
      fprintf(file,"Perc %.61f", 100.0 *
                                 (psatom->stat[c][y]).writmisscnt /
                                 (psatom->stat[c][y]).writcnt);
    fprintf(file,"\n         TOTAL %12lu ",
                                 (psatom->stat[c][y]).instcnt +
                                 (psatom->stat[c][y]).readcnt +
                                 (psatom->stat[c][y]).writcnt);
    fprintf(file,"Miss %12lu ", (psatom->stat[c][y]).instmisscnt +
                                (psatom->stat[c][y]).readmisscnt +
                                (psatom->stat[c][y]).writmisscnt);
    if (((psatom->stat[c][y]).instcnt +
         (psatom->stat[c][y]).readcnt +
         (psatom->stat[c][y]).writcnt) != 0)
      fprintf(file,"Perc %.61f", 100.0 *
                                 ((psatom->stat[c][y]).instmisscnt +
                                  (psatom->stat[c][y]).readmisscnt +
                                  (psatom->stat[c][y]).writmisscnt) /
                                 ((psatom->stat[c][y]).instcnt +
                                  (psatom->stat[c][y]).readcnt +
                                  (psatom->stat[c][y]).writcnt));
    fprintf(file,"\n        Int (times process %d overwrote:)\n", y);
    for (z=0; z <= psatom->numtasks; z++)
      fprintf(file,"               Process %d = %12lu\n", z,
                                 (psatom->stat[c][y]).interfere[z]);
    fprintf(file,"             (process %d is invalid data)\n",
                                              psatom->numtasks);
    }
/* PRINT TOTAL CACHE PERFORMANCE */
fprintf(file,"     ********************\n");
fprintf(file,"     TOTAL FOR CACHE\n");
fprintf(file,"         Inst  %12lu ", total.instcnt);
fprintf(file,"Miss %12lu ", total.instmisscnt);
if (total.instcnt != 0)
  fprintf(file,"Perc %.61f", 100.0 * total.instmisscnt /
```

```
                                        total.instcnt);
  fprintf(file,"\n         Data  %12lu ", total.readcnt +
                                           total.writcnt);
  fprintf(file,"Miss %12lu ", total.readmisscnt + total.writmisscnt);
  if ((total.readcnt + total.writcnt) != 0)
    fprintf(file,"Perc %.6lf", 100.0 *
                               (total.readmisscnt + total.writmisscnt)/
                               (total.readcnt + total.writcnt));
  fprintf(file,"\n          read %12lu ", total.readcnt);
  fprintf(file,"Miss %12lu ", total.readmisscnt);
  if (total.readcnt != 0)
    fprintf(file,"Perc %.6lf", 100.0 * total.readmisscnt /
                                        total.readcnt);
  fprintf(file,"\n          writ %12lu ", total.writcnt);
  fprintf(file,"Miss %12lu ", total.writmisscnt);
  if (total.writcnt != 0)
    fprintf(file,"Perc %.6lf", 100.0 * total.writmisscnt /
                                        total.writcnt);
  fprintf(file,"\n         TOTAL %12lu ", total.instcnt +
                                           total.readcnt +
                                           total.writcnt);
  fprintf(file,"Miss %12lu ", total.instmisscnt +
                              total.readmisscnt +
                              total.writmisscnt);
  if ((total.instcnt + total.readcnt + total.writcnt) != 0)
    fprintf(file,"Perc %.6lf", 100.0 * (total.instmisscnt +
                                        total.readmisscnt +
                                        total.writmisscnt) /
                                       (total.instcnt +
                                        total.readcnt +
                                        total.writcnt));
  fprintf(file,"\n");
  fprintf(file,"\f");
  }
fclose(file);
/* IF LAST PROCESS, SHUT DOWN SIMULATION */
psatom->count--;
if (psatom->count == 0)
  {
  psatom->numcaches = 0;
  psatom->numtasks = 0;
  }
return;
}
```

B Tables of Simulation Results

Key to data tables:

Miss Data

- Inst = instruction fetch misses
- Read = data read misses
- Write = data write misses
- Data = total data read and write misses
- Total = total misses
- % = miss rate

Interference Data (Int(#))

- Process 0 is the kernel, except for simulations with the context switch model where process 0 is the test program.
- Additional process' numbers are shown in the same order as the tables.
- The extra process is for cases where invalid data is overwritten (at simulation start).

B.1 Compress Alone

B.2 GCC Alone

B.3 Espresso Alone

B.4 Alvinn Alone

B.5 Compress w/ Operating System

B.6 GCC w/ Operating System

B.7 Espresso w/ Operating System

B.8 Alvinn w/ Operating System

B.9 Compress and GCC w/ Operating System

B.10 Compress and Espresso w/ Operating System

B.11 GCC and Espresso w/ Operating System

B.12 Compress w/ Model, n=1

B.13 GCC w/ Model, n=1

B.14 Espresso w/ Model, n=1

B.15 Alvinn w/ Model, n=1

B.16 Compress w/ Model, n=2

B.17 GCC w/ Model, n=2

B.18 Espresso w/ Model, n=2

Espresso data: Table 40

Reference Statistics:															
Total Instruction References			87045943												
Data Reads			22412017												
Data writes			8521660												
Total Data References			30933677												
Total References			117979620												
Miss Statistics:															
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)	
0	548488	0.6301	3969460	17.7113	78951	0.9265	4048411	13.0874	4596899	3.8964	4596771	128			
1	135306	0.1554	3576626	15.9585	47558	0.5581	3624184	11.7160	3759490	3.1866	3759234	256			
2	64726	0.0744	3247221	14.4887	39424	0.4626	3286645	10.6248	3351371	2.8406	3350859	512			
3	9242	0.0106	2929900	13.0729	18561	0.2178	2948461	9.5316	2957703	2.5070	2957191	512			
4	1506638	1.7309	4424039	19.7396	195294	2.2917	4619333	14.9330	6125971	5.1924	6125720	251			
5	1328	0.0015	3985445	17.7826	98415	1.1549	4083860	13.2020	4085188	3.4626	4084932	256			
6	517	0.0006	3858349	17.2155	78860	0.9254	3937209	12.7279	3937726	3.3376	3937470	256			
7	1004580	1.1541	4740845	21.1531	264522	3.1041	5005367	16.1810	6009947	5.0941	6009820	127			
8	924	0.0011	4337294	19.3525	129378	1.5182	4466672	14.4395	4467596	3.7868	4467468	128			
9	368	0.0004	4027651	17.9709	75480	0.8857	4103131	13.2643	4103499	3.4781	4103371	128			
10	1004399	1.1539	5615207	25.0544	318116	3.7330	5933323	19.1808	6937722	5.8804	6937658	64			
11	1164	0.0013	4994496	22.2849	177969	2.0884	5172465	16.7211	5173629	4.3852	5173565	64			
12	369	0.0004	4355536	19.4339	114656	1.3455	4470192	14.4509	4470561	3.7893	4470497	64			
13	933	0.0011	3953631	17.6407	134243	1.5753	4087874	13.2150	4088807	3.4657	4088330	477			
14	665	0.0008	3659598	16.3287	77320	0.9073	3736918	12.0804	3737583	3.1680	3737090	493			
15	431	0.0005	3596518	16.0473	70934	0.8324	3667452	11.8559	3667883	3.1089	3667374	509			
16	752	0.0009	4174166	18.6247	153044	1.7959	4327210	13.9887	4327962	3.6684	4327716	246			
17	538	0.0006	3784252	16.8849	60460	0.7095	3844712	12.4289	3845250	3.2592	3844999	251			
18	281	0.0003	3714918	16.5756	43175	0.5067	3758093	12.1489	3758374	3.1856	3758118	256			
19	508	0.0006	4503993	20.0963	181579	2.1308	4685572	15.1472	4686080	3.9719	4685953	127			
20	462	0.0005	4025134	17.9597	75802	0.8895	4100936	13.2572	4101398	3.4764	4101271	127			
21	175	0.0002	3897335	17.3895	43030	0.5049	3940365	12.7381	3940540	3.3400	3940412	128			
22	655	0.0008	3593087	16.0320	99735	1.1704	3692822	11.9379	3693477	3.1306	3692647	830			
23	419	0.0005	3392960	15.1390	71452	0.8385	3464412	11.1995	3464831	2.9368	2463962	869			
24	420	0.0005	3352182	14.9571	69680	0.8177	3421862	11.0619	3422282	2.9007	3421402	880			
25	500	0.0006	3753498	16.7477	90768	1.0651	3844266	12.4274	3844766	3.2588	3844329	437			
26	263	0.0003	3505179	15.6397	41908	0.4918	3547087	11.4667	3547350	3.0067	3546889	461			
27	264	0.0003	3459600	15.4364	36153	0.4242	3495753	11.3008	3496017	2.9632	3495549	468			
28	283	0.0003	3994077	17.8211	93747	1.1001	4087824	13.2148	4088107	3.4651	4087878	229			
29	160	0.0002	3635400	16.2208	34495	0.4048	3669895	11.8638	3670055	3.1108	3669814	241			
30	164	0.0002	3572923	15.9420	22919	0.2689	3595842	11.6244	3596006	3.0480	3595762	244			
31	499	0.0006	3336377	14.8866	63903	0.7499	3400280	10.9922	3400779	2.8825	3400013	766			
32	262	0.0003	3200179	14.2789	36587	0.4293	3236766	10.4636	3237028	2.7437	3236258	770			
33	262	0.0003	3163806	14.1166	34970	0.4104	3198776	10.3408	3199038	2.7115	3198264	774			
34	279	0.0003	3479719	15.5261	57598	0.6759	3537317	11.4352	3537596	2.9985	3537189	407			
35	157	0.0002	3319045	14.8092	21947	0.2575	3340992	10.8005	3341149	2.8320	3340740	409			
36	157	0.0002	3276713	14.6203	18121	0.2126	3294834	10.6513	3294991	2.7928	3294578	413			
37	218	0.0003	3642992	16.2546	80614	0.9460	3723606	12.0374	3723824	3.1563	3723606	218			
38	96	0.0001	3431770	15.3122	23850	0.2799	3455620	11.1711	3455716	2.9291	3455496	220			
39	96	0.0001	3376695	15.0664	13679	0.1605	3390374	10.9601	3390470	2.8738	3390247	223			

Table 6: Compress Alone

Reference Statistics:														
Total Instruction References			160240141											
Data Reads			50197329											
Data writes			19074844											
Total Data References			69272173											
Total References			229512314											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	5634791	3.5165	3899643	7.7686	1124204	5.8936	5023847	7.2523	10658638	4.6440	10658510	128		
1	3467582	2.1640	2294130	4.5702	625965	3.2816	2920095	4.2154	6387677	2.7832	6387421	256		
2	1812638	1.1312	1199783	2.3901	292406	1.5329	1492189	2.1541	3304827	1.4399	3304315	512		
3	698472	0.4359	570583	1.1367	93526	0.4903	664109	0.9587	1362581	0.5937	1362069	512		
4	7952135	4.9626	5068732	10.0976	1850587	9.7017	6919319	9.9886	14871454	6.4796	14871198	256		
5	7609862	4.7490	3384080	6.7416	1246657	6.5356	4630737	6.6848	12240599	5.3333	12240343	256		
6	7475162	4.6650	2831492	5.6407	1082598	5.6755	3914090	5.6503	11389252	4.9624	11388996	256		
7	5980164	3.7320	5790039	11.5346	1688672	8.8529	7478711	10.7961	13458875	5.8641	13458747	128		
8	5825266	3.6353	3760599	7.4916	1005669	5.2722	4766268	6.8805	10591534	4.6148	10591406	128		
9	5705809	3.5608	3076405	6.1286	858898	4.5028	3935303	5.6809	9641112	4.2007	9640984	128		
10	4487462	2.8005	6883193	13.7123	1671740	8.7641	8554933	12.3497	13042395	5.6827	13042331	64		
11	4385553	2.7369	4483221	8.9312	913221	4.7876	5396442	7.7902	9781995	4.2621	9781931	64		
12	4275364	2.6681	3483294	6.9392	745730	3.9095	4229024	6.1049	8504388	3.7054	8504324	64		
13	5266989	3.2869	3159463	6.2941	1191492	6.2464	4350955	6.2810	9617944	4.1906	9617432	512		
14	4673839	2.9168	1976429	3.9373	777475	4.0759	2753904	3.9755	7427743	3.2363	7427231	512		
15	4373242	2.7292	1528259	3.0445	638520	3.3474	2166779	3.1279	6540021	2.8495	6539509	512		
16	4114016	2.5674	3584271	7.1404	1083978	5.6828	4668249	6.7390	8782265	3.8265	8782009	256		
17	3767259	2.3510	2181824	4.3465	591549	3.1012	2773373	4.0036	6540632	2.8498	6540376	256		
18	3601147	2.2473	1637423	3.2620	443952	2.3274	2081375	3.0046	5682522	2.4759	5682266	256		
19	3225225	2.0127	4211466	8.3898	1025053	5.3738	5236519	7.5593	8461744	3.6868	8461616	128		
20	3021710	1.8857	2449200	4.8791	513529	2.6922	2962729	4.2769	5984439	2.6075	5984311	128		
21	2954404	1.8437	1916946	3.8188	350334	1.8366	2267280	3.2730	5221684	2.2751	5221556	128		
22	3405512	2.1253	1846283	3.6781	616452	3.2318	2462735	3.5552	5868247	2.5568	5867223	1024		
23	2336318	1.4580	1082124	2.1557	411736	2.1585	1493860	2.1565	3830178	1.6688	3829154	1024		
24	1827906	1.1407	850798	1.6949	370338	1.9415	1221136	1.7628	3049042	1.3285	3048018	1024		
25	2707757	1.6898	1949160	3.8830	511914	2.6837	2461074	3.5528	5168831	2.2521	5168319	512		
26	1937995	1.2094	1077259	2.1460	290744	1.5242	1368003	1.9748	3305998	1.4404	3305486	512		
27	1608192	1.0036	809191	1.6120	235986	1.2372	1045177	1.5088	2653369	1.1561	2652857	512		
28	2163484	1.3502	2211021	4.4047	473562	2.4827	2684583	3.8754	4848067	2.1123	4847811	256		
29	1670226	1.0423	1202397	2.3953	229876	1.2051	1432273	2.0676	3102499	1.3518	3102243	256		
30	1496380	0.9338	839524	1.6724	171024	0.8966	1010548	1.4588	2506928	1.0923	2506672	256		
31	1333373	0.8321	1109603	2.2105	299395	1.5696	1408998	2.0340	2742371	1.1949	2741347	1024		
32	957751	0.5977	486658	0.9695	117426	0.6156	604084	0.8720	1561835	0.6805	1560811	1024		
33	645400	0.4028	373960	0.7450	82256	0.4312	456216	0.6586	1101616	0.4800	1100592	1024		
34	1064758	0.6645	1234739	2.4598	279605	1.4658	1514344	2.1861	2579102	1.1237	2578590	512		
35	789890	0.4929	495222	0.9866	92268	0.4837	587490	0.8481	1377380	0.6001	1376868	512		
36	573265	0.3578	350341	0.6979	60173	0.3155	410514	0.5926	983779	0.4286	983267	512		
37	892309	0.5569	1494708	2.9777	311513	1.6331	1806221	2.6074	2698530	1.1758	2698274	256		
38	663995	0.4144	585821	1.1670	82571	0.4329	668392	0.9649	1332387	0.5805	1332131	256		
39	523915	0.3270	371607	0.7403	51510	0.2700	423117	0.6108	947032	0.4126	946776	256		

Table 7: GCC Alone

Reference Statistics:														
Total Instruction References			977787923											
Data Reads			225779346											
Data writes			59867420											
Total Data References			285646766											
Total References			1263434689											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	8828519	0.9029	11703074	5.1834	2373219	3.9641	14076293	4.9279	22904812	1.8129	22904684	128		
1	4865847	0.4976	6069567	2.6883	1490980	2.4905	7560547	2.6468	12426394	0.9835	12426138	256		
2	2220854	0.2271	2579074	1.1423	879209	1.4686	3458283	1.2107	5679137	0.4495	5678625	512		
3	377680	0.0386	660418	0.2925	148988	0.2489	809406	0.2834	1187086	0.0940	1186574	512		
4	14693581	1.5027	23118162	10.2393	3936506	6.5754	27054668	9.4714	41748249	3.3043	41747993	256		
5	9252564	0.9463	15990947	7.0826	2836005	4.7371	18826952	6.5910	28079516	2.2225	28079260	256		
6	8053922	0.8237	13308022	5.8943	2723865	4.5498	16031887	5.6125	24085809	1.9064	24085553	256		
7	10452313	1.0690	23397494	10.3630	3415200	5.7046	26812694	9.3867	37265007	2.9495	37264879	128		
8	6526085	0.6674	14083232	6.2376	2256025	3.7684	16339257	5.7201	22865342	1.8098	22865214	128		
9	5649109	0.5777	11217249	4.9682	2129752	3.5574	13347001	4.6726	18996110	1.5035	18995982	128		
10	8490767	0.8684	27228681	12.0599	3403098	5.6844	30631779	10.7237	39122546	3.0965	39122482	64		
11	5102567	0.5218	15491933	6.8615	2033721	3.3970	17525654	6.1354	22628221	1.7910	22628157	64		
12	4629917	0.4735	12549858	5.5585	1834442	3.0642	14384300	5.0357	19014217	1.5050	19014153	64		
13	9658262	0.9878	15250352	6.7545	2634907	4.4012	17885259	6.2613	27543521	2.1801	27543009	512		
14	4163585	0.4258	8917113	3.9495	1934506	3.2313	10851619	3.7990	15015204	1.1884	15014692	512		
15	1928044	0.1972	8115711	3.5945	1599530	2.6718	9715241	3.4011	11643285	0.9216	11642773	512		
16	7148710	0.7311	14689495	6.5061	2154180	3.5983	16843675	5.8967	23992385	1.8990	23992129	256		
17	3010235	0.3079	6967032	3.0858	1421812	2.3749	8388844	2.9368	11399079	0.9022	11398823	256		
18	1379080	0.1410	5648439	2.5018	1091308	1.8229	6739747	2.3595	8118827	0.6426	8118571	256		
19	5920445	0.6055	16422816	7.2738	2042168	3.4112	18464984	6.4643	24385429	1.9301	24385301	128		
20	2521273	0.2579	6727026	2.9795	1215816	2.0308	7942842	2.7807	10464115	0.8282	10463987	128		
21	1279054	0.1308	5032940	2.2291	861735	1.4394	5894675	2.0636	7173729	0.5678	7173601	128		
22	3221746	0.3295	8141361	3.6059	1770436	2.9573	9911797	3.4699	13133543	1.0395	13132523	1020		
23	871639	0.0891	3263216	1.4453	1118778	1.8688	4381994	1.5341	5253633	0.4158	5252609	1024		
24	238537	0.0244	2263067	1.0023	923179	1.5420	3186246	1.1154	3424783	0.2711	3423759	1024		
25	2089245	0.2137	7785487	3.4483	1382636	2.3095	9168123	3.2096	11257368	0.8910	11256857	511		
26	578122	0.0591	2792241	1.2367	772999	1.2912	3565240	1.2481	4143362	0.3279	4142850	512		
27	181469	0.0186	1655168	0.7331	568692	0.9499	2223860	0.7785	2405329	0.1904	2404817	512		
28	1454824	0.1488	8755641	3.8780	1301227	2.1735	10056868	3.5207	11511692	0.9111	11511436	256		
29	390129	0.0399	2792620	1.2369	614879	1.0271	3407499	1.1929	3797628	0.3006	3797372	256		
30	169147	0.0173	1405648	0.6226	393630	0.6575	1799278	0.6299	1968425	0.1558	1968169	256		
31	174801	0.0179	3530287	1.5636	608814	1.0169	4139101	1.4490	4313902	0.3414	4312895	1007		
32	73537	0.0075	1056081	0.4686	328197	0.5482	1386278	0.4853	1459815	0.1155	1458794	1021		
33	11858	0.0012	269080	0.1192	162279	0.2711	431359	0.1510	443217	0.0351	442193	1024		
34	132836	0.0136	3882047	1.7194	531877	0.8884	4413924	1.5452	4546760	0.3599	4546251	509		
35	59872	0.0061	1125507	0.4985	263212	0.4397	1388719	0.4862	1448591	0.1147	1448079	512		
36	15465	0.0016	262666	0.1163	115398	0.1928	378064	0.1324	393529	0.0311	393017	512		
37	122332	0.0125	5096588	2.2573	644486	1.0765	5741074	2.0099	5863406	0.4641	5863150	256		
38	51647	0.0053	1849075	0.8190	250909	0.4191	2099984	0.7352	2151631	0.1703	2151375	256		
39	21193	0.0022	348444	0.1543	94040	0.1571	442484	0.1549	463677	0.0367	463421	256		

Table 8: Espresso Alone

Reference Statistics:														
Total Instruction References			5233222111											
Data Reads			1415013652											
Data writes			487428474											
Total Data References			1902442126											
Total References			7135664237											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	11005331	0.2103	54138053	3.8260	1174196	0.2409	55312249	2.9074	66317580	0.9294	66317452	128		
1	6022427	0.1151	34388743	2.4303	618355	0.1269	35007098	1.8401	41029525	0.5750	41029269	256		
2	1208323	0.0231	30807454	2.1772	155641	0.0319	30963095	1.6275	32171418	0.4509	32170906	512		
3	302266	0.0058	15165827	1.0718	66787	0.0137	15232614	0.8007	15534880	0.2177	15534368	512		
4	13058392	0.2495	144132672	10.1860	1941066	0.3982	146073738	7.6782	159132130	2.2301	159131874	256		
5	12949146	0.2474	122906539	8.6859	1226573	0.2516	124133112	6.5249	137082258	1.9211	137082002	256		
6	14135066	0.2701	116881920	8.2601	1345725	0.2761	118227645	6.2145	132362711	1.8549	132362455	256		
7	9786658	0.1870	115400206	8.1554	1962273	0.4026	117362479	6.1690	127149137	1.7819	127149009	128		
8	9973793	0.1906	72036989	5.0909	1088521	0.2233	73125510	3.8438	83099303	1.1646	83099175	128		
9	10288822	0.1966	65887136	4.6563	946660	0.1942	66833796	3.5131	77122618	1.0808	77122490	128		
10	7000911	0.1338	129734746	9.1684	2153297	0.4418	131888043	6.9326	138888954	1.9464	138888890	64		
11	6825446	0.1304	52969960	3.7434	1215446	0.2494	54185406	2.8482	61010852	0.8550	61010788	64		
12	6901854	0.1319	48396579	3.4202	782898	0.1606	49179477	2.5851	56081331	0.7859	56081267	64		
13	8062348	0.1541	105356026	7.4456	1313935	0.2696	106669961	5.6070	114732309	1.6079	114731798	511		
14	7078832	0.1353	88868170	6.2804	1122743	0.2303	89990913	4.7303	97069745	1.3603	97069233	512		
15	3265578	0.0624	100813045	7.1245	878793	0.1803	101691838	5.3453	104957416	1.4709	104956904	512		
16	6190225	0.1183	76152635	5.3818	1172519	0.2406	77325154	4.0645	83515379	1.1704	83515123	256		
17	6191430	0.1183	49112663	3.4708	794038	0.1629	49906701	2.6233	56098131	0.7862	56097875	256		
18	5093199	0.0973	55142574	3.8970	630158	0.1293	55772732	2.9316	60865931	0.8530	60865675	256		
19	4628400	0.0884	76749969	5.4240	1438910	0.2952	78188879	4.1099	82817279	1.1606	82817151	128		
20	4787100	0.0915	31063288	2.1953	945155	0.1939	32008443	1.6825	36795543	0.5157	36795415	128		
21	4553096	0.0870	33652826	2.3783	501501	0.1029	34154327	1.7953	38707423	0.5425	38707295	128		
22	5147360	0.0984	81394706	5.7522	767669	0.1575	82162375	4.3188	87309735	1.2236	87308803	932		
23	2826646	0.0540	61784739	4.3664	422837	0.0867	62207576	3.2699	65034222	0.9114	65033243	979		
24	796115	0.0152	60829173	4.2988	196052	0.0402	61025225	3.2077	61821340	0.8664	61820341	999		
25	4218157	0.0806	50702077	3.5832	750200	0.1539	51452277	2.7045	55670434	0.7802	55669950	484		
26	2871097	0.0549	31749540	2.2438	438421	0.0899	32187961	1.6919	35059058	0.4913	35058557	501		
27	531347	0.0102	30734026	2.1720	118828	0.0244	30852854	1.6217	31384201	0.4398	31383694	507		
28	3102588	0.0593	42148644	2.9787	1057388	0.2169	43206032	2.2711	46308620	0.6490	46308373	247		
29	2271429	0.0434	16968240	1.1992	339889	0.0697	17308129	0.9098	19579558	0.2744	19579306	252		
30	1408558	0.0269	15989489	1.1300	69568	0.0143	16059057	0.8441	17467615	0.2448	17467359	256		
31	1477563	0.0282	38531712	2.7231	529506	0.1086	39061218	2.0532	40538781	0.5681	40537925	856		
32	233167	0.0045	30402183	2.1485	144186	0.0296	30546369	1.6056	30779536	0.4313	30778661	875		
33	451	0.0000	30031166	2.1223	99439	0.0204	30130605	1.5838	30131056	0.4223	30130139	917		
34	1175119	0.0225	26698673	1.8868	698739	0.1434	27397412	1.4401	28572531	0.4004	28572082	449		
35	569059	0.0109	15501358	1.0955	86501	0.0177	15587859	0.8194	16156918	0.2264	16156459	459		
36	274	0.0000	15076517	1.0655	52846	0.0108	15129363	0.7953	15129637	0.2120	15129154	483		
37	890481	0.0170	33990717	2.4021	946894	0.1943	34937611	1.8365	35828092	0.5021	35827854	238		
38	697880	0.0133	8404184	0.5939	86084	0.0177	8490268	0.4463	9188148	0.1288	9187906	242		
39	196	0.0000	7663506	0.5416	44727	0.0092	7708233	0.4052	7708429	0.1080	7708178	251		

Table 9: Alvinn Alone

Table 10: Compress w/ Operating System, Compress Data

Reference Statistics:														
Total Instruction References			87045969											
Data Reads			22412010											
Data writes			8521661											
Total Data References			30933671											
Total References			117979640											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	Int(0)	Int(1)	Int(2)	Int(3)
0	1025051	1.1776	4546006	20.2838	132145	1.5507	4678151	15.1232	5703202	4.8341	537959	5165239	4	
1	623679	0.7165	4142362	18.4828	77742	0.9123	4220104	13.6424	4843783	4.1056	446116	4397663	4	
2	67593	0.0777	3301508	14.7310	41380	0.4856	3342888	10.8066	3410481	2.8907	308475	3102002	4	
3	10311	0.0118	2995041	13.3636	21010	0.2465	3016051	9.7501	3026362	2.5652	251970	2774388	4	
4	1595489	1.8329	4971786	22.1836	220431	2.5867	5192217	16.7850	6787706	5.7533	458148	6329552	6	
5	99364	0.1142	4172066	18.6153	116601	1.3683	4288667	13.8641	4388031	3.7193	467269	3920756	6	
6	30419	0.0349	3923112	17.5045	83315	0.9777	4006427	12.9517	4036846	3.4216	411901	3624939	6	
7	1070135	1.2294	5441962	24.2815	464169	5.4469	5906131	19.0929	6976266	5.9131	379254	6597008	4	
8	70127	0.0806	5011443	22.3605	202187	2.3726	5213630	16.8542	5283757	4.4785	414062	4869691	4	
9	26051	0.0299	4203069	18.7536	113691	1.3341	4316760	13.9549	4342811	3.6810	382464	3960343	4	
10	1057406	1.2148	7712673	34.4131	555842	6.5227	8268515	26.7298	9325921	7.9047	291315	9034602	4	
11	75586	0.0868	6203676	27.6801	327130	3.8388	6530806	21.1123	6606392	5.5996	338682	6267706	4	
12	39921	0.0459	4743503	21.1650	241717	2.8365	4985220	16.1158	5025141	4.2593	344276	4680861	4	
13	82225	0.0945	4510650	20.1260	155451	1.8242	4666101	15.0842	4748326	4.0247	391799	4356521	6	
14	46508	0.0534	3733556	16.6587	85694	1.0056	3819250	12.3466	3865758	3.2766	345536	3520216	6	
15	9551	0.0110	3641835	16.2495	72630	0.8523	3714465	12.0078	3724016	3.1565	317865	3406145	6	
16	58771	0.0675	4919384	21.9498	376754	4.4211	5296138	17.1209	5354909	4.5388	340015	5014890	4	
17	42505	0.0488	3912600	17.4576	113525	1.3322	4026125	13.0153	4068630	3.4486	352098	3716528	4	
18	7201	0.0083	3809735	16.9986	53960	0.6332	3863695	12.4903	3870896	3.2810	337018	3533874	4	
19	56619	0.0650	6697255	29.8824	424659	4.9833	7121914	23.0232	7178533	6.0846	318023	6860506	4	
20	39634	0.0455	4623117	20.6279	181126	2.1255	4804243	15.5308	4843877	4.1057	331979	4511894	4	
21	5104	0.0059	4163923	18.5790	94288	1.1065	4258211	13.7656	4263315	3.6136	347857	3915454	4	
22	7184	0.0083	3633167	16.2108	100813	1.1830	3733980	12.0709	3741164	3.1710	271575	3469681	8	
23	6663	0.0077	3446798	15.3792	76131	0.8934	3522929	11.3887	3529592	2.9917	256957	3272629	6	
24	5935	0.0068	3386659	15.1109	70396	0.8261	3457055	11.1757	3462990	2.9352	227529	3235455	6	
25	5150	0.0059	3790361	16.9122	90849	1.0661	3881210	12.5469	3886360	3.2941	263802	3622554	4	
26	4799	0.0055	3585067	15.9962	69536	0.8160	3654603	11.8143	3659402	3.1017	268627	3390771	4	
27	4816	0.0055	3491876	15.5804	38491	0.4517	3530367	11.4127	3535183	2.9964	251259	3283920	4	
28	3665	0.0042	4043477	18.0416	125958	1.4781	4169435	13.4786	4173100	3.5371	283279	3889817	4	
29	3875	0.0045	3777649	16.8555	95575	1.1216	3873224	12.5211	3877099	3.2862	288856	3588239	4	
30	4116	0.0047	3647456	16.2746	33567	0.3939	3681023	11.8997	3685139	3.1235	304770	3380365	4	
31	3953	0.0045	3369811	15.0357	64261	0.7541	3434072	11.1014	3438025	2.9141	211503	3226513	9	
32	2464	0.0028	3233512	14.4276	37792	0.4435	3271304	10.5752	3273768	2.7749	191745	3082016	7	
33	1713	0.0020	3199548	14.2760	35698	0.4189	3235246	10.4587	3236959	2.7437	179581	3057374	4	
34	2860	0.0033	3524497	15.7259	72172	0.8469	3596669	11.6270	3599529	3.0510	232591	3366934	4	
35	1811	0.0021	3356383	14.9758	27483	0.3225	3383866	10.9391	3385677	2.8697	231177	3154496	4	
36	1356	0.0016	3317306	14.8015	21238	0.2492	3338544	10.7926	3339900	2.8309	227402	3112494	4	
37	2117	0.0024	3684337	16.4391	83660	0.9817	3767997	12.1809	3770114	3.1956	232668	3537442	4	
38	1472	0.0017	3467971	15.4737	28443	0.3338	3496414	11.3029	3497886	2.9648	256781	3241101	4	
39	1234	0.0014	3421706	15.2673	20208	0.2371	3441914	11.1268	3443148	2.9184	268756	3174388	4	

Table 11: Compress w/ Operating System, Operating System Data

Reference Statistics:														
Total Instruction References			5567602											
Data Reads			1518924											
Data writes			802242											
Total Data References			2321166											
Total References			7888768											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	405077	7.2756	442622	29.1405	93040	11.5975	535662	23.0773	940739	11.9250	402680	537935	124	
1	312348	5.6101	337407	22.2136	76603	9.5486	414010	17.8363	726358	9.2075	280030	446076	252	
2	206867	3.7155	202735	13.3473	59230	7.3831	261965	11.2859	468832	5.9430	160013	308311	508	
3	129180	2.3202	165278	10.8813	36173	4.5090	201451	8.6789	330631	4.1912	78379	251744	508	
4	357511	6.4213	493350	32.4802	149469	18.6314	642819	27.6938	1000330	12.6804	541979	458101	250	
5	354136	6.3607	467378	30.7703	146837	18.3033	614215	26.4615	968351	12.2751	500868	467233	250	
6	290358	5.2151	460057	30.2883	146567	18.2697	606624	26.1345	896982	11.3704	484864	411868	250	
7	257137	4.6185	531977	35.0233	102472	12.7732	634449	27.3332	891586	11.3020	512236	379226	124	
8	253271	4.5490	459508	30.2522	97131	12.1074	556639	23.9810	809910	10.2666	395749	414037	124	
9	209728	3.7669	454015	29.8906	96921	12.0813	550936	23.7353	760664	9.6424	378098	382442	124	
10	196456	3.5286	565678	37.2420	100294	12.5017	665972	28.6913	862428	10.9324	571062	291306	60	
11	220249	3.9559	457075	30.0920	81094	10.1084	538169	23.1853	758418	9.6139	419686	338672	60	
12	188872	3.3923	451543	29.7278	80279	10.0068	531822	22.9118	720694	9.1357	376368	344266	60	
13	271541	4.8772	416370	27.4122	130866	16.3125	547236	23.5759	818777	10.3790	426603	391668	506	
14	211089	3.7914	358928	23.6304	122246	15.2380	481174	20.7298	692263	8.7753	346328	345429	506	
15	177930	3.1958	339954	22.3812	122259	15.2397	462213	19.9130	640143	8.1146	321865	317772	506	
16	194474	3.4930	477050	31.4071	89596	11.1682	566646	24.4121	761120	9.6481	420910	339958	252	
17	157375	2.8266	379047	24.9550	82758	10.3158	461805	19.8954	619180	7.8489	266877	352051	252	
18	129757	2.3306	376366	24.7785	81642	10.1767	458008	19.7318	587765	7.4507	250536	336977	252	
19	156779	2.8159	478023	31.4712	86839	10.8245	564862	24.3353	721641	9.1477	403510	318007	124	
20	121869	2.1889	400732	26.3826	69131	8.6172	469863	20.2425	591732	7.5009	259644	331964	124	
21	98562	1.7703	392650	25.8505	68234	8.5054	460884	19.8557	559446	7.0917	211480	347842	124	
22	145989	2.6221	301130	19.8252	110704	13.7993	411834	17.7425	557823	7.0711	285593	271214	1016	
23	125510	2.2543	264957	17.4437	101996	12.7139	366953	15.8090	492463	6.2426	234839	256606	1018	
24	113474	2.0381	223478	14.7129	97270	12.1248	320748	13.8184	434222	5.5043	206021	227183	1018	
25	103705	1.8627	310966	20.4728	74069	9.2328	385035	16.5880	488740	6.1954	224585	263647	508	
26	91884	1.6503	293582	19.3283	68380	8.5236	361962	15.5940	453846	5.7531	184853	268485	508	
27	83808	1.5053	263797	17.3674	65629	8.1807	329426	14.1923	413234	5.2383	161602	251124	508	
28	78935	1.4178	340717	22.4315	62125	7.7439	402842	17.3552	481777	6.1071	198301	283224	252	
29	69701	1.2519	339906	22.3781	57262	7.1377	397168	17.1107	466869	5.9181	177808	288809	252	
30	64229	1.1536	322741	21.2480	55665	6.9387	378406	16.3024	442635	5.6110	137647	304736	252	
31	68293	1.2266	238917	15.7294	63877	7.9623	302794	13.0449	371087	4.7040	158977	211095	1015	
32	55661	0.9997	185372	12.2042	55064	6.8638	240436	10.3584	296097	3.7534	103734	191346	1017	
33	47824	0.8590	168891	11.1191	51506	6.4203	220397	9.4951	268221	3.4000	88022	179179	1020	
34	50939	0.9149	276257	18.1877	52241	6.5119	328498	14.1523	379437	4.8098	156510	232419	508	
35	42911	0.7707	230417	15.1698	44929	5.6004	275346	11.8624	318257	4.0343	86733	231016	508	
36	37029	0.6651	226637	14.9209	42020	5.2378	268657	11.5742	305686	3.8750	77933	227245	508	
37	41449	0.7445	307787	20.2635	41599	5.1853	349386	15.0522	390835	4.9543	157982	232601	252	
38	34515	0.6199	285045	18.7662	36495	4.5491	321540	13.8525	356055	4.5134	99079	256724	252	
39	30050	0.5397	280815	18.4878	34475	4.2973	315290	13.5833	345340	4.3776	76379	268709	252	

Reference Statistics:														
Total Instruction References			92613571											
Data Reads			23930934											
Data writes			9323903											
Total Data References			33254837											
Total References			125868408											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	1430128	1.5442	4988628	20.8459	225185	2.4151	5213813	15.6784	6643941	5.2785				
1	936027	1.0107	4479769	18.7196	154345	1.6554	4634114	13.9352	5570141	4.4254				
2	274460	0.2963	3504243	14.6432	100610	1.0791	3604853	10.8401	3879313	3.0820				
3	139491	0.1506	3160319	13.2060	57183	0.6133	3217502	9.6753	3356993	2.6671				
4	1953000	2.1088	5465136	22.8371	369900	3.9672	5835036	17.5464	7788036	6.1874				
5	453500	0.4897	4639444	19.3868	263438	2.8254	4902882	14.7434	5356382	4.2555				
6	320777	0.3464	4383169	18.3159	229882	2.4655	4613051	13.8718	4933828	3.9198				
7	1327272	1.4331	5973939	24.9633	566641	6.0773	6540580	19.6681	7867852	6.2509				
8	323398	0.3492	5470951	22.8614	299318	3.2102	5770269	17.3517	6093667	4.8413				
9	235779	0.2546	4657084	19.4605	210612	2.2588	4867696	14.6376	5103475	4.0546				
10	1253862	1.3539	8278351	34.5927	656136	7.0371	8934487	26.8667	10188349	8.0944				
11	295835	0.3194	6660751	27.8332	408224	4.3783	7068975	21.2570	7364810	5.8512				
12	228793	0.2470	5195046	21.7085	321996	3.4534	5517042	16.5902	5745835	4.5650				
13	353766	0.3820	4927020	20.5885	286317	3.0708	5213337	15.6769	5567103	4.4230				
14	257597	0.2781	4092484	17.1012	207940	2.2302	4300424	12.9317	4558021	3.6213				
15	187481	0.2024	3981789	16.6387	194889	2.0902	4176678	12.5596	4364159	3.4672				
16	253245	0.2734	5396434	22.5500	466350	5.0017	5862784	17.6299	6116029	4.8591				
17	199880	0.2158	4291647	17.9335	196283	2.1052	4487930	13.4956	4687810	3.7244				
18	136958	0.1479	4186101	17.4924	135602	1.4543	4321703	12.9957	4458661	3.5423				
19	213398	0.2304	7175278	29.9833	511498	5.4859	7686776	23.1148	7900174	6.2765				
20	161503	0.1744	5023849	20.9931	250257	2.6840	5274106	15.8597	5435609	4.3185				
21	103666	0.1119	4556573	19.0405	162522	1.7431	4719095	14.1907	4822761	3.8316				
22	153173	0.1654	3934297	16.4402	211517	2.2685	4145814	12.4668	4298987	3.4155				
23	132173	0.1427	3711755	15.5103	178127	1.9104	3889882	11.6972	4022055	3.1954				
24	119409	0.1289	3610137	15.0857	167666	1.7982	3777803	11.3602	3897212	3.0963				
25	108855	0.1175	4101327	17.1382	164918	1.7688	4266245	12.8289	4375100	3.4759				
26	96683	0.1044	3878649	16.2077	137916	1.4792	4016565	12.0781	4113248	3.2679				
27	88624	0.0957	3755673	15.6938	104120	1.1167	3859793	11.6067	3948417	3.1369				
28	82600	0.0892	4384194	18.3202	188083	2.0172	4572277	13.7492	4654877	3.6982				
29	73576	0.0794	4117555	17.2060	152837	1.6392	4270392	12.8414	4343968	3.4512				
30	68345	0.0738	3970197	16.5902	89232	0.9570	4059429	12.2070	4127774	3.2794				
31	72246	0.0780	3608728	15.0798	128138	1.3743	3736866	11.2371	3809112	3.0263				
32	58125	0.0628	3418884	14.2865	92856	0.9959	3511740	10.5601	3569865	2.8362				
33	49537	0.0535	3368439	14.0757	87204	0.9353	3455643	10.3914	3505180	2.7848				
34	53799	0.0581	3800754	15.8822	124413	1.3343	3925167	11.8033	3978966	3.1612				
35	44722	0.0483	3586800	14.9881	72412	0.7766	3659212	11.0035	3703934	2.9427				
36	38385	0.0414	3543943	14.8090	63258	0.6784	3607201	10.8471	3645586	2.8963				
37	43566	0.0470	3992124	16.6819	125259	1.3434	4117383	12.3813	4160949	3.3058				
38	35987	0.0389	3753016	15.6827	64938	0.6965	3817954	11.4809	3853941	3.0619				
39	31284	0.0338	3702521	15.4717	54683	0.5865	3757204	11.2982	3788488	3.0099				

Table 12: Compress w/ Operating System, Combined Data

Reference Statistics:														
Total Instruction References			160240175											
Data Reads			50197333											
Data writes			19074845											
Total Data References			69272178											
Total References			229512353											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	5930477	3.7010	4052214	8.0726	1185136	6.2131	5237350	7.5605	11167827	4.8659	1289475	9878349	3	
1	3822004	2.3852	2571230	5.1222	720183	3.7756	3291413	4.7514	7113417	3.0994	1012010	6101401	3	
2	2113535	1.3190	1395532	2.7801	346031	1.8141	1741563	2.5141	3855098	1.6797	609264	3245831	3	
3	859011	0.5361	715612	1.4256	115348	0.6047	830960	1.1996	1689971	0.7363	362672	1327296	3	
4	8256812	5.1528	5126873	10.2134	1911328	10.0201	7038201	10.1602	15295013	6.6641	1487260	13807749	4	
5	7936576	4.9529	3657769	7.2868	1359035	7.1247	5016804	7.2422	12953380	5.6439	1551400	11401976	4	
6	7795851	4.8651	3150933	6.2771	1180574	6.1892	4331507	6.2529	12127358	5.2840	1560222	10567132	4	
7	6192677	3.8646	5821084	11.5964	1702487	8.9253	7523571	10.8609	13716248	5.9763	1167505	12548740	3	
8	6030794	3.7636	3975395	7.9195	1111031	5.8246	5086426	7.3427	11117220	4.8438	1291563	9825654	3	
9	5910602	3.6886	3401159	6.7756	943898	4.9484	4345057	6.2724	10255659	4.4685	1321784	8933872	3	
10	4637520	2.8941	6917919	13.7814	1651305	8.6570	8569224	12.3704	13206744	5.7543	892643	12314098	3	
11	4519059	2.8202	4616766	9.1972	1018907	5.3416	5635673	8.1356	10154732	4.4245	1011166	9143563	3	
12	4401698	2.7469	3762245	7.4949	815985	4.2778	4578230	6.6090	8979928	3.9126	1119597	7860328	3	
13	5606294	3.4987	3489105	6.9508	1325313	6.9480	4814418	6.9500	10420712	4.5404	1175931	9244777	4	
14	5029798	3.1389	2168464	4.3199	837043	4.3882	3005507	4.3387	8035305	3.5010	1064529	6970772	4	
15	4750077	2.9643	1753549	3.4933	697878	3.6586	2451427	3.5388	7201504	3.1377	965750	6235750	4	
16	4351960	2.7159	3834987	7.6398	1142031	5.9871	4977018	7.1847	9328978	4.0647	955489	8373486	3	
17	3997278	2.4946	2343766	4.6691	634558	3.3267	2978324	4.2995	6975602	3.0393	984131	5991468	3	
18	3858935	2.4082	1894655	3.7744	505538	2.6503	2400193	3.4649	6259128	2.7271	952200	5306925	3	
19	3398843	2.1211	4546950	9.0582	1092878	5.7294	5639828	8.1415	9038671	3.9382	838561	8200107	3	
20	3182775	1.9863	2617399	5.2142	557327	2.9218	3174726	4.5830	6357501	2.7700	924426	5433072	3	
21	3134727	1.9563	2211580	4.4058	430779	2.2584	2642359	3.8145	5777086	2.5171	988080	4789003	3	
22	3711553	2.3162	2079376	4.1424	700333	3.6715	2779709	4.0127	6491262	2.8283	896069	5595187	6	
23	2632041	1.6426	1273761	2.5375	487227	2.5543	1760988	2.5421	4393029	1.9141	669575	3723450	4	
24	2109715	1.3166	988193	1.9686	435087	2.2809	1423280	2.0546	3532995	1.5393	517052	3015939	4	
25	2933355	1.8306	2260666	4.5036	608653	3.1909	2869319	4.1421	5802674	2.5283	757813	5044858	3	
26	2162131	1.3493	1300811	2.5914	353783	1.8547	1654594	2.3885	3816725	1.6630	615412	3201310	3	
27	1849125	1.1540	974296	1.9409	284265	1.4903	1258561	1.8168	3107686	1.3540	493709	2613974	3	
28	2337676	1.4589	2609680	5.1988	562915	2.9511	3172595	4.5799	5510271	2.4009	716434	4793834	3	
29	1844020	1.1508	1490190	2.9687	292088	1.5313	1782278	2.5729	3626298	1.5800	622311	3003984	3	
30	1697127	1.0591	1088659	2.1688	220885	1.1580	1309544	1.8904	3006671	1.3100	596125	2410543	3	
31	1520994	0.9492	1231307	2.4529	332304	1.7421	1563611	2.2572	3084605	1.3440	548709	2535884	12	
32	1118374	0.6979	611281	1.2178	147792	0.7748	759073	1.0958	1877447	0.8180	356033	1521408	6	
33	758014	0.4730	471089	0.9385	112037	0.5874	583126	0.8418	1341140	0.5843	256632	1084505	3	
34	1208988	0.7545	1379626	2.7484	302024	1.5834	1681650	2.4276	2890638	1.2595	505048	2385587	3	
35	928693	0.5796	662791	1.3204	118571	0.6216	781362	1.1280	1710055	0.7451	354779	1355273	3	
36	679583	0.4241	491159	0.9785	90694	0.4755	581853	0.8400	1261436	0.5496	254029	1007404	3	
37	1006019	0.6278	1650462	3.2879	331914	1.7401	1982376	2.8617	2988395	1.3021	475721	2512671	3	
38	776098	0.4843	801293	1.5963	109206	0.5725	910499	1.3144	1686597	0.7349	406270	1280324	3	
39	623706	0.3892	593204	1.1817	84226	0.4416	677430	0.9779	1301136	0.5669	327276	973857	3	

Table 13: GCC w/ Operating System, GCC Data

Table 14: GCC w/ Operating System, Operating System Data

Reference Statistics:														
Total Instruction References			18705569											
Data Reads			5130601											
Data writes			2613506											
Total Data References			7744107											
Total References			26449676											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	1563695	8.3595	1305087	25.4373	298668	11.4279	1603755	20.7094	3167450	11.9754	1877866	1289459	125	
1	1261469	6.7438	950376	18.5237	228719	8.7514	1179095	15.2257	2440564	9.2272	1428319	1011992	253	
2	920733	4.9222	565185	11.0160	150281	5.7502	715466	9.2388	1636199	6.1861	1026454	609236	509	
3	475427	2.5416	378429	7.3759	76164	2.9142	454593	5.8702	930020	3.5162	566859	362652	509	
4	2249643	12.0266	1358536	26.4791	417673	15.9813	1776209	22.9363	4025852	15.2208	2538367	1487233	252	
5	2208587	11.8071	1211623	23.6156	391479	14.9791	1603102	20.7009	3811689	14.4111	2260062	1551375	252	
6	2229488	11.9188	1205169	23.4898	383152	14.6605	1588321	20.5101	3817809	14.4342	2257362	1560195	252	
7	1566655	8.3753	1498433	29.2058	307981	11.7842	1806414	23.3263	3373069	12.7528	2205451	1167493	125	
8	1558716	8.3329	1262423	24.6058	275544	10.5431	1537967	19.8598	3096683	11.7078	1805008	1291550	125	
9	1562454	8.3529	1238467	24.1388	269649	10.3175	1508116	19.4744	3070570	11.6091	1748678	1321767	125	
10	1141680	6.1034	1625907	31.6904	299528	11.4608	1925435	24.8632	3067115	11.5960	2174415	892639	61	
11	1139865	6.0937	1314548	25.6217	236133	9.0351	1550681	20.0240	2690546	10.1723	1679323	1011162	61	
12	1140816	6.0988	1285294	25.0515	223986	8.5703	1509280	19.4894	2650096	10.0194	1530443	1119592	61	
13	1896999	10.1414	1055232	20.5674	329326	12.6009	1384558	17.8789	3281557	12.4068	2105157	1175892	508	
14	1721879	9.2052	827001	16.1190	276721	10.5881	1103722	14.2524	2825601	10.6829	1760609	1064484	508	
15	1650888	8.8256	698692	13.6181	264933	10.1371	963625	12.4433	2614513	9.8849	1648298	965707	508	
16	1329781	7.1090	1210514	23.5940	239222	9.1533	1449736	18.7205	2779517	10.5087	1823796	955468	253	
17	1223010	6.5382	902120	17.5831	202929	7.7646	1105049	14.2695	2328059	8.8018	1343704	984102	253	
18	1192607	6.3757	838998	16.3528	193503	7.4040	1032501	13.3327	2225108	8.4126	1272674	952181	253	
19	970521	5.1884	1277557	24.9007	227816	8.7169	1505373	19.4389	2475894	9.3608	1637217	838552	125	
20	912963	4.8807	1059232	20.6454	171530	6.5632	1230762	15.8929	2143725	8.1049	1219186	924414	125	
21	899815	4.8104	999887	19.4887	172015	6.5818	1171902	15.1328	2071717	7.8327	1083521	988071	125	
22	1524924	8.1522	721889	14.0703	238250	9.1161	960139	12.3983	2485063	9.3954	1588025	896020	1018	
23	1316126	7.0360	524038	10.2140	199563	7.6358	723601	9.3439	2039727	7.7117	1369184	669523	1020	
24	1176729	6.2908	385567	7.5150	176854	6.7669	562421	7.2626	1739150	6.5753	1221133	516997	1020	
25	1072103	5.7315	741561	14.4537	172649	6.6060	914210	11.8052	1986313	7.5098	1228020	757784	509	
26	928018	4.9612	575839	11.2236	145355	5.5617	721194	9.3128	1649212	6.2353	1033322	615381	509	
27	836111	4.4699	441008	8.5956	123685	4.7325	564693	7.2919	1400804	5.2961	906618	493677	509	
28	790683	4.2270	810257	15.7926	144278	5.5205	954535	12.3260	1745218	6.5983	1028546	716419	253	
29	690587	3.6919	718984	14.0136	115549	4.4212	834533	10.7764	1525120	5.7661	902573	622294	253	
30	630461	3.3704	624259	12.1674	100299	3.8377	724558	9.3562	1355019	5.1230	758657	596109	253	
31	795328	4.2518	530467	10.3393	136898	5.2381	667365	8.6177	1462693	5.5301	913012	548669	1012	
32	684331	3.6584	304534	5.9356	92843	3.5524	397377	5.1313	1081708	4.0897	724693	355997	1018	
33	611305	3.2680	229632	4.4757	80049	3.0629	309681	3.9989	920986	3.4820	663370	256595	1021	
34	584869	3.1267	594961	11.5963	113509	4.3432	708470	9.1485	1293339	4.8898	787803	505027	509	
35	507120	2.7111	353350	6.8871	68628	2.6259	421978	5.4490	929098	3.5127	573829	354760	509	
36	440994	2.3576	281629	5.4892	57743	2.2094	339372	4.3823	780366	2.9504	525849	254008	509	
37	449358	2.4023	684496	13.3414	102910	3.9376	787406	10.1678	1236764	4.6759	760801	475710	253	
38	384100	2.0534	532251	10.3740	61866	2.3672	594117	7.6719	978217	3.6984	571705	406259	253	
39	333050	1.7805	424489	8.2737	50080	1.9162	474569	6.1281	807619	3.0534	480102	327264	253	

Table 15: GCC w/ Operating System, Combined Data

Reference Statistics:																
Total Instruction References	178945744															
Data Reads	55327934															
Data writes	21688351															
Total Data References	77016285															
Total References	255962029															
Miss Statistics:																
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)		
0	7494172	4.1880	5357301	9.6828	1483804	6.8415	6841105	8.8827	14335277	5.6005						
1	5083473	2.8408	3521606	6.3650	948902	4.3752	4470508	5.8046	9553981	3.7326						
2	3034268	1.6956	1960717	3.5438	496312	2.2884	2457029	3.1903	5491297	2.1454						
3	1334438	0.7457	1094041	1.9774	191512	0.8830	1285553	1.6692	2619991	1.0236						
4	10506455	5.8713	6485409	11.7218	2329001	10.7385	8814410	11.4449	19320865	7.5483						
5	10145163	5.6694	4869392	8.8010	1750514	8.0712	6619906	8.5955	16765069	6.5498						
6	10025339	5.6024	4356102	7.8732	1563726	7.2100	5919828	7.6865	15945167	6.2295						
7	7759332	4.3361	7319517	13.2293	2010468	9.2698	9329985	12.1143	17089317	6.6765						
8	7589510	4.2412	5237818	9.4669	1366575	6.3932	6624393	8.6013	14213903	5.5531						
9	7473056	4.1762	4639626	8.3857	1213547	5.5954	5853173	7.5999	13326229	5.2063						
10	5779200	3.2296	8543826	15.4422	1950833	8.9948	10494659	13.6265	16273859	6.3579						
11	5658924	3.1624	5931314	10.7203	1255040	5.7867	7186354	9.3310	12845278	5.0184						
12	5542514	3.0973	5047539	9.1229	1039971	4.7951	6087510	7.9042	11630024	4.5437						
13	7503293	4.1931	4544337	8.2135	1654639	7.6292	6198976	8.0489	13702269	5.3532						
14	6751677	3.7730	2995465	5.4140	1113764	5.1353	4109229	5.3355	10860906	4.2432						
15	6400965	3.5770	2452241	4.4322	962811	4.4393	3415052	4.4342	9816017	3.8350						
16	5681741	3.1751	5045501	9.1193	1381253	6.3686	6426754	8.3447	12108495	4.7306						
17	5220288	2.9172	3245886	5.8666	837487	3.8615	4083373	5.3020	9303661	3.6348						
18	5051542	2.8229	2733653	4.9408	699041	3.2231	3432694	4.4571	8484236	3.3146						
19	4369364	2.4417	5824507	10.5272	1320694	6.0894	7145201	9.2775	11514565	4.4985						
20	4095738	2.2888	3676631	6.6452	728857	3.3606	4405488	5.7202	8501226	3.3213						
21	4034542	2.2546	3211467	5.8044	602794	2.7793	3814261	4.9525	7848803	3.0664						
22	5236477	2.9263	2801265	5.0630	938583	4.3276	3739848	4.8559	8976325	3.5069						
23	3948167	2.2063	1797799	3.2494	686790	3.1666	2484589	3.2261	6432756	2.5132						
24	3286444	1.8366	1373760	2.4829	611941	2.8215	1985701	2.5783	5272145	2.0597						
25	4005458	2.2384	3002227	5.4262	781302	3.6024	3783529	4.9126	7788987	3.0430						
26	3090149	1.7269	1876650	3.3919	499138	2.3014	2375788	3.0848	5465937	2.1354						
27	2685236	1.5006	1415304	2.5580	407950	1.8810	1823254	2.3674	4508490	1.7614						
28	3128359	1.7482	3419937	6.1812	707193	3.2607	4127130	5.3588	7255489	2.8346						
29	2534607	1.4164	2209174	3.9929	407637	1.8795	2616811	3.3977	5151418	2.0126						
30	2327588	1.3007	1712918	3.0959	321184	1.4809	2034102	2.6411	4361690	1.7040						
31	2316322	1.2944	1761774	3.1842	469202	2.1634	2230976	2.8968	4547298	1.7766						
32	1802705	1.0074	915815	1.6552	240635	1.1095	1156450	1.5016	2959155	1.1561						
33	1369319	0.7652	700721	1.2665	192086	0.8857	892807	1.1592	2262126	0.8838						
34	1793857	1.0025	1974587	3.5689	415533	1.9159	2390120	3.1034	4183977	1.6346						
35	1435813	0.8024	1016141	1.8366	187199	0.8631	1203340	1.5624	2639153	1.0311						
36	1120577	0.6262	772788	1.3967	148437	0.6844	921225	1.1961	2041802	0.7977						
37	1455377	0.8133	2334958	4.2202	434824	2.0049	2769782	3.5964	4225159	1.6507						
38	1160198	0.6484	1333544	2.4103	171072	0.7888	1504616	1.9536	2664814	1.0411						
39	956756	0.5347	1017693	1.8394	134306	0.6193	1151999	1.4958	2108755	0.8239						

Table 16: Espresso w/ Operating System, Espresso Data

Reference Statistics:														
Total Instruction References			977787899											
Data Reads			225779331											
Data writes			59867421											
Total Data References			285646752											
Total References			1263434651											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	9569442	0.9787	12946056	5.7339	2610631	4.3607	15556687	5.4461	25126129	1.9887	3817682	21308442	5	
1	5304178	0.5425	7458893	3.3036	1781552	2.9758	9240445	3.2349	14544623	1.1512	2173271	12371347	5	
2	2319886	0.2373	3558705	1.5762	1186363	1.9817	4745068	1.6612	7064954	0.5592	1054237	6010712	5	
3	590974	0.0604	1253188	0.5550	261853	0.4374	1515041	0.5304	2106015	0.1667	508482	1597529	4	
4	15753707	1.6112	24140719	10.6922	4209396	7.0312	28350115	9.9249	44103822	3.4908	4085718	40018097	7	
5	10607427	1.0848	17402955	7.7079	3246266	5.4224	20649221	7.2289	31256648	2.4739	4380885	26875756	7	
6	9536900	0.9754	14412412	6.3834	2999708	5.0106	17412120	6.0957	26949020	2.1330	4230257	22718756	7	
7	11248371	1.1504	24122303	10.6840	3678136	6.1438	27800439	9.7325	39048810	3.0907	3319712	35729093	5	
8	7453695	0.7623	15590899	6.9054	2607510	4.3555	18198409	6.3709	25652104	2.0303	3842762	21809337	5	
9	6595231	0.6745	12270033	5.4345	2329653	3.8914	14599686	5.1111	21194917	1.6776	3873572	17321340	5	
10	9086613	0.9293	27860405	12.3397	3579030	5.9783	31439435	11.0064	40526048	3.2076	2594963	37931081	4	
11	5736277	0.5867	17053144	7.5530	2284046	3.8152	19337190	6.7696	25073467	1.9845	3166669	21906794	4	
12	5246911	0.5366	13513250	5.9852	1978815	3.3053	15492065	5.4235	20738976	1.6415	3626193	17112779	4	
13	10410726	1.0647	16753236	7.4202	3114155	5.2018	19867391	6.9552	30276117	2.3965	2695683	27582427	7	
14	5016652	0.5131	10126012	4.4849	2234461	3.7323	12360473	4.3272	17377125	1.3754	2343112	15034006	7	
15	2793692	0.2857	9159016	4.0566	1831506	3.0593	10990522	3.8476	13784214	1.0910	1918679	11865528	7	
16	7722999	0.7898	16278865	7.2101	2660925	4.4447	18939790	6.6305	26662789	2.1103	2267929	24394855	5	
17	3652766	0.3736	8275477	3.6653	1730636	2.8908	10006113	3.5030	13658879	1.0811	2316051	11342823	5	
18	2111793	0.2160	6927674	3.0683	1346647	2.2494	8274321	2.8967	10386114	0.8221	2064016	8322093	5	
19	6401207	0.6547	18038289	7.9893	2482666	4.1469	20520955	7.1840	26922162	2.1309	2077491	24844667	4	
20	3078831	0.3149	8017334	3.5510	1445044	2.4137	9462378	3.3126	12541209	0.9926	2276886	10264319	4	
21	1974422	0.2019	6353820	2.8142	1102316	1.8413	7456136	2.6103	9430558	0.7464	2523066	6907488	4	
22	3669931	0.3753	9847205	4.3614	2319410	3.8742	12166615	4.2593	15836546	1.2535	1768802	14067735	9	
23	1258195	0.1287	4412383	1.9543	1422312	2.3758	5834695	2.0426	7092890	0.5614	1197903	5894979	8	
24	506621	0.0518	3151965	1.3960	1109763	1.8537	4261728	1.4920	4768349	0.3774	702820	4065520	9	
25	2429849	0.2485	9652200	4.2751	2004618	3.3484	11656818	4.0809	14086667	1.1150	1552607	12534055	5	
26	892803	0.0913	4038413	1.7887	1080039	1.8041	5118452	1.7919	6011255	0.4758	1194758	4816492	5	
27	394198	0.0403	2609915	1.1560	747705	1.2489	3357620	1.1754	3751818	0.2970	754453	2997360	5	
28	1744288	0.1784	10847047	4.8043	1854144	3.0971	12701191	4.4465	14445479	1.1433	1541546	12903929	4	
29	679444	0.0695	4108770	1.8198	867005	1.4482	4975775	1.7419	5655219	0.4476	1233405	4421810	4	
30	353139	0.0361	2458029	1.0887	574069	0.9589	3032098	1.0615	3385237	0.2679	1025880	2359353	4	
31	398195	0.0407	4694718	2.0793	868144	1.4501	5562862	1.9475	5961057	0.4718	1081226	4879822	9	
32	222485	0.0228	1969718	0.8724	660029	1.1025	2629747	0.9206	2852232	0.2258	616444	2235781	7	
33	52103	0.0053	572867	0.2537	279850	0.4674	852717	0.2985	904820	0.0716	226490	678323	7	
34	284635	0.0291	5193764	2.3004	826110	1.3799	6019874	2.1075	6304509	0.4990	1041888	5262616	5	
35	166830	0.0171	2143082	0.9492	522381	0.8726	2665463	0.9331	2832293	0.2242	652221	2180068	4	
36	52667	0.0054	672626	0.2979	224966	0.3758	897592	0.3142	950259	0.0752	280526	669729	4	
37	263601	0.0270	6566647	2.9084	962236	1.6073	7528883	2.6357	7792484	0.6168	1011811	6780669	4	
38	170088	0.0174	2805531	1.2426	524484	0.8761	3330015	1.1658	3500103	0.2770	767670	2732429	4	
39	59067	0.0060	941089	0.4168	218157	0.3644	1159246	0.4058	1218313	0.0964	427854	790455	4	

Table 17: Espresso w/ Operating System, Operating System Data

Reference Statistics:																
Total Instruction References	29093428															
Data Reads	9107479															
Data writes	3585537															
Total Data References	12693016															
Total References	41786444															
Miss Statistics:																
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	Int(0)	Int(1)	Int(2)	Int(3)		
0	2266718	7.7912	2836510	31.1448	392197	10.9383	3228707	25.4369	5495425	13.1512	1677662	3817640	123			
1	1280716	4.4021	1543092	16.9431	327692	9.1393	1870784	14.7387	3151500	7.5419	978029	2173220	251			
2	585695	2.0132	616465	6.7688	278932	7.7794	895397	7.0542	1481092	3.5444	426422	1054163	507			
3	184769	0.6351	401079	4.4038	166002	4.6298	567081	4.4677	751850	1.7993	242957	508385	508			
4	2291526	7.8764	2872197	31.5367	611117	17.0439	3483314	27.4428	5774840	13.8199	1688931	4085660	249			
5	2447757	8.4134	2707461	29.7279	560701	15.6379	3268162	25.7477	5715919	13.6789	1334850	4380820	249			
6	2380794	8.1833	2767110	30.3828	510100	14.2266	3277210	25.8190	5658004	13.5403	1427563	4230192	249			
7	1705289	5.8614	3500484	38.4353	461322	12.8662	3961806	31.2125	5667095	13.5620	2347296	3319676	123			
8	1887001	6.4860	3019789	33.1572	421151	11.7458	3440940	27.1089	5327941	12.7504	1485101	3842717	123			
9	1876162	6.4487	2977246	32.6901	395698	11.0359	3372944	26.5732	5249106	12.5617	1375458	3873525	123			
10	1352894	4.6502	4148213	45.5473	578053	16.1218	4726266	37.2352	6079160	14.5482	3484155	2594945	60			
11	1527836	5.2515	3325923	36.5186	439048	12.2450	3764971	29.6618	5292807	12.6663	2126101	3166646	60			
12	1545051	5.3107	3402292	37.3571	413955	11.5451	3816247	30.0657	5361298	12.8302	1735070	3626168	60			
13	1399372	4.8099	2109320	23.1603	527410	14.7094	2636730	20.7731	4036102	9.6589	1340014	2695583	505			
14	1147585	3.9445	1611756	17.6971	371824	10.3701	1983580	15.6273	3131165	7.4933	787629	2343031	505			
15	1015164	3.4893	1326765	14.5679	296752	8.2764	1623517	12.7906	2638681	6.3147	719572	1918604	505			
16	1054348	3.6240	2778763	30.5108	403381	11.2502	3182144	25.0700	4236492	10.1384	1968362	2267879	251			
17	887714	3.0513	1848438	20.2958	320615	8.9419	2169053	17.0886	3056767	7.3152	740516	2316000	251			
18	876625	3.0131	1714211	18.8220	267809	7.4691	1982020	15.6150	2858645	6.8411	794429	2063965	251			
19	861531	2.9613	3057520	33.5715	500076	13.9470	3557596	28.0280	4419127	10.5755	2341535	2077468	124			
20	754732	2.5942	2450628	26.9079	337074	9.4009	2787702	21.9625	3542434	8.4775	1265454	2276856	124			
21	828884	2.8490	2489201	27.3314	305893	8.5313	2795094	22.0207	3623978	8.6726	1100822	2523032	124			
22	830442	2.8544	1276991	14.0213	448007	12.4948	1724998	13.5901	2555440	6.1155	785901	1768524	1015			
23	570528	1.9610	865869	9.7268	276123	7.7010	1161992	9.1546	1732520	4.1461	533841	1197663	1016			
24	412257	1.4170	542059	5.9518	158140	4.4105	700199	5.5164	1112456	2.6622	408778	702663	1015			
25	614151	2.1110	1277627	14.0283	350210	9.7673	1627837	12.8247	2241988	5.3653	688984	1552497	507			
26	456770	1.5700	976416	10.7210	263573	7.3510	1239989	9.7691	1696759	4.0605	501592	1194660	507			
27	305356	1.0496	694627	7.6270	143389	3.9991	838016	6.6022	1143372	2.7362	388476	754389	507			
28	494059	1.6982	1463964	16.0743	350860	9.7854	1814824	14.2978	2308883	5.5254	767123	1541508	252			
29	375780	1.2916	1389328	15.2548	284380	7.9313	1673708	13.1861	2049488	4.9047	815874	1233362	252			
30	245684	0.8445	1132844	12.4386	168923	4.7112	1301767	10.2558	1547451	3.7032	521355	1025844	252			
31	401812	1.3811	935544	10.2723	295691	8.2468	1231235	9.7001	1633047	3.9081	551136	1080896	1015			
32	242081	0.8321	406406	4.4623	222706	6.2112	629112	4.9564	871193	2.0849	254013	616163	1017			
33	123239	0.4236	225160	2.4723	76892	2.1445	302052	2.3797	425291	1.0178	198046	226228	1017			
34	283848	0.9756	1136616	12.4800	296991	8.2830	1433607	11.2945	1717455	4.1101	675185	1041763	507			
35	179060	0.6155	473331	5.1972	251637	7.0181	724968	5.7116	904028	2.1634	251409	652111	508			
36	99211	0.3410	289524	3.1790	96182	2.6825	385706	3.0387	484917	1.1605	203974	280435	508			
37	248650	0.8547	1422437	15.6183	255729	7.1322	1678166	13.2212	1926816	4.6111	914797	1011767	252			
38	172615	0.5933	925889	10.1662	224144	6.2513	1150033	9.0604	1322648	3.1653	554758	767638	252			
39	89232	0.3067	566138	6.2381	115729	3.2277	683867	5.3877	773099	1.8501	345026	427821	252			

Table 18: Espresso w/ Operating System, Combined Data

Reference Statistics:														
Total Instruction References			1006881327											
Data Reads			234886810											
Data writes			63452958											
Total Data References			298339768											
Total References			1305221095											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	11836160	1.1755	15782566	6.7192	3002828	4.7324	18785394	6.2966	30621554	2.3461				
1	6584894	0.6540	9001985	3.8325	2109244	3.3241	11111229	3.7244	17696123	1.3558				
2	2905581	0.2886	4175170	1.7775	1465295	2.3093	5640465	1.8906	8546046	0.6548				
3	775743	0.0770	1654267	0.7043	427855	0.6743	2082122	0.6979	2857865	0.2190				
4	18045233	1.7922	27012916	11.5004	4820513	7.5970	31833429	10.6702	49878662	3.8215				
5	13055184	1.2966	20110416	8.5617	3806967	5.9997	23917383	8.0168	36972567	2.8327				
6	11917694	1.1836	17179522	7.3140	3509808	5.5314	20689330	6.9348	32607024	2.4982				
7	12953660	1.2865	27622787	11.7600	4139458	6.5237	31762245	10.6463	44715905	3.4259				
8	9340696	0.9277	18610688	7.9233	3028661	4.7731	21639349	7.2533	30980045	2.3735				
9	8471393	0.8413	15247279	6.4913	2725351	4.2951	17972630	6.0242	26444023	2.0260				
10	10439507	1.0368	32008618	13.6273	4157083	6.5514	36165701	12.1223	46605208	3.5707				
11	7264113	0.7214	20379067	8.6761	2723094	4.2915	23102161	7.7436	30366274	2.3265				
12	6791962	0.6746	16915542	7.2016	2392770	3.7709	19308312	6.4719	26100274	1.9997				
13	11810098	1.1729	18862556	8.0305	3641565	5.7390	22504121	7.5431	34314219	2.6290				
14	6164237	0.6122	11737768	4.9972	2606285	4.1074	14344053	4.8080	20508290	1.5713				
15	3808856	0.3783	10485781	4.4642	2128258	3.3541	12614039	4.2281	16422895	1.2582				
16	8777347	0.8717	19057628	8.1135	3064306	4.8293	22121934	7.4150	30899281	2.3674				
17	4540480	0.4509	10123915	4.3101	2051251	3.2327	12175166	4.0810	16715646	1.2807				
18	2988418	0.2968	8641885	3.6792	1614456	2.5443	10256341	3.4378	13244759	1.0148				
19	7262738	0.7213	21095809	8.9813	2982742	4.7007	24078551	8.0708	31341289	2.4012				
20	3833563	0.3807	10467962	4.4566	1782118	2.8086	12250080	4.1061	16083643	1.2323				
21	2803306	0.2784	8843021	3.7648	1408209	2.2193	10251230	3.4361	13054536	1.0002				
22	4500373	0.4470	11124196	4.7360	2767417	4.3614	13891613	4.6563	18391986	1.4091				
23	1828723	0.1816	5298252	2.2557	1698435	2.6767	6996687	2.3452	8825410	0.6762				
24	918878	0.0913	3694024	1.5727	1267903	1.9982	4961927	1.6632	5880805	0.4506				
25	3044000	0.3023	10929827	4.6532	2354828	3.7111	13284655	4.4529	16328655	1.2510				
26	1349573	0.1340	5014829	2.1350	1343612	2.1175	6358441	2.1313	7708014	0.5906				
27	699554	0.0695	3304542	1.4069	891094	1.4043	4195636	1.4063	4895190	0.3750				
28	2238347	0.2223	12311011	5.2413	2205004	3.4750	14516015	4.8656	16754362	1.2836				
29	1055224	0.1048	5498098	2.3407	1151385	1.8145	6649483	2.2288	7704707	0.5903				
30	598823	0.0595	3590873	1.5288	742992	1.1709	4333865	1.4527	4932688	0.3779				
31	800007	0.0795	5630262	2.3970	1163835	1.8342	6794097	2.2773	7594104	0.5818				
32	464566	0.0461	2376124	1.0116	882735	1.3912	3258859	1.0923	3723425	0.2853				
33	175342	0.0174	798027	0.3397	356742	0.5622	1154769	0.3871	1330111	0.1019				
34	568483	0.0565	6330380	2.6951	1123101	1.7700	7453481	2.4983	8021964	0.6146				
35	345890	0.0344	2616413	1.1139	774018	1.2198	3390431	1.1364	3736321	0.2863				
36	151878	0.0151	962150	0.4096	321148	0.5061	1283298	0.4301	1435176	0.1100				
37	512251	0.0509	7989084	3.4012	1217965	1.9195	9207049	3.0861	9719300	0.7446				
38	342703	0.0340	3731420	1.5886	748628	1.1798	4480048	1.5017	4822751	0.3695				
39	148299	0.0147	1509227	0.6425	333886	0.5262	1843113	0.6178	1991412	0.1526				

Table 19: Alvinn w/ Operating System, Alvinn Data

Reference Statistics:																					
Total Instruction References		5233222045																			
Data Reads		1415013630																			
Data writes		487428474																			
Total Data References		1902442104																			
Total References		7135664149																			
Miss Statistics:																					
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)							
0	11777917	0.2251	62467434	4.4146	1108083	0.2273	63575517	3.3418	75353434	1.0560	24787556	50565874		4							
1	6578977	0.1257	42371342	2.9944	797167	0.1635	43168509	2.2691	49747486	0.6972	13896363	35851119		4							
2	1924125	0.0368	34077484	2.4083	435766	0.0894	34513250	1.8142	36437375	0.5106	6207819	30229552		4							
3	725741	0.0139	16136411	1.1405	199569	0.0409	16337980	0.8588	17063721	0.2391	2674896	14388822		3							
4	14878361	0.2843	146170073	10.3299	2185383	0.4483	148355456	7.7982	163238817	2.2876	22531639	140702172		6							
5	14513801	0.2773	123772438	8.7471	1162620	0.2385	124935058	6.5671	139448859	1.9543	23385504	116063349		6							
6	14843070	0.2836	117197316	8.2824	1107420	0.2272	118304736	6.2186	133147806	1.8659	21158637	111989163		6							
7	11171914	0.2135	117935199	8.3346	2430479	0.4986	120365678	6.3269	131537592	1.8434	17634903	113902685		4							
8	10961997	0.2095	73735639	5.2109	1235948	0.2536	74971587	3.9408	85933584	1.2043	19632409	66301171		4							
9	10737140	0.2052	67221068	4.7506	833766	0.1711	68054834	3.5772	78791974	1.1042	18150035	60641935		4							
10	8489032	0.1622	132232267	9.3449	2794657	0.5733	135026924	7.0976	143515956	2.0112	13359982	130155971		3							
11	7653268	0.1462	55448795	3.9186	1201134	0.2464	56649929	2.9777	64303197	0.9012	15097123	49206071		3							
12	7271857	0.1390	50151550	3.5442	1056811	0.2168	51208361	2.6917	58480218	0.8195	16739279	41740936		3							
13	9680485	0.1850	110011838	7.7746	1500417	0.3078	111512255	5.8615	121192740	1.6984	17432768	103759966		6							
14	7859999	0.1502	97020567	6.8565	943596	0.1936	97964163	5.1494	105824162	1.4830	15629299	90194857		6							
15	4494747	0.0859	108202653	7.6468	886558	0.1819	109089211	5.7342	113583958	1.5918	12606336	100977616		6							
16	7234715	0.1382	80266622	5.6725	1685488	0.3458	81952110	4.3077	89186825	1.2499	13721219	75465602		4							
17	6748159	0.1289	55379753	3.9137	812097	0.1666	56191850	2.9537	62940009	0.8820	13962924	48977081		4							
18	5876298	0.1123	60121189	4.2488	659410	0.1353	60780599	3.1949	66656897	0.9341	11563076	55093817		4							
19	5562092	0.1063	81058358	5.7285	1758443	0.3608	82816801	4.3532	88378893	1.2386	11522410	76856480		3							
20	5186282	0.0991	35669431	2.5208	686790	0.1409	36356221	1.9110	41542503	0.5822	12386569	29155931		3							
21	5019936	0.0959	37390873	2.6424	581716	0.1193	37972589	1.9960	42992525	0.6025	11678686	31113836		3							
22	6695064	0.1279	85109423	6.0147	1266602	0.2599	86376025	4.5403	93071089	1.3043	12406282	80664799		8							
23	3300869	0.0631	67410120	4.7639	646562	0.1326	68056682	3.5773	71357551	1.0000	8232595	63124950		6							
24	1069862	0.0204	63641732	4.4976	412744	0.0847	64054476	3.3670	65124338	0.9127	4441509	60682823		6							
25	5215696	0.0997	53909781	3.8098	1416145	0.2905	55325926	2.9082	60541622	0.8484	10102977	50438641		4							
26	3261854	0.0623	36387738	2.5715	607147	0.1246	36994885	1.9446	40256739	0.5642	7583898	32672837		4							
27	1047012	0.0200	34017385	2.4040	371287	0.0762	34388672	1.8076	35435684	0.4966	4156254	31279426		4							
28	3986302	0.0762	45432561	3.2108	1569047	0.3219	47001608	2.4706	50987910	0.7146	8990014	41997893		3							
29	2542594	0.0486	21084869	1.4901	497351	0.1020	21582220	1.1344	24124814	0.3381	6888218	17236593		3							
30	1709106	0.0327	19655828	1.3891	392097	0.0804	20047925	1.0538	21757031	0.3049	5185574	16571454		3							
31	2467927	0.0472	40443243	2.8582	854394	0.1753	41297637	2.1708	43765564	0.6133	6673606	37091941		17							
32	537486	0.0103	32125862	2.2704	411882	0.0845	32537744	1.7103	33075230	0.4635	3213824	29861401		5							
33	20180	0.0004	30338982	2.1441	173243	0.0355	30512225	1.6038	30532405	0.4279	1407103	29125298		4							
34	2052415	0.0392	28461976	2.0114	811917	0.1666	29273893	1.5388	31326308	0.4390	5929902	25396402		4							
35	784018	0.0150	17141520	1.2114	445411	0.0914	17586931	0.9244	18370949	0.2575	2901456	15469490		3							
36	140147	0.0027	15773499	1.1147	245530	0.0504	16019029	0.8420	16159176	0.2265	1471488	14687685		3							
37	1544996	0.0295	35821055	2.5315	1064827	0.2185	36885882	1.9389	38430878	0.5386	5140263	33290612		3							
38	867060	0.0166	10367353	0.7327	457435	0.0938	10824788	0.5690	11691848	0.1639	3328570	8363275		3							
39	203514	0.0039	8813128	0.6228	259603	0.0533	9072731	0.4769	9276245	0.1300	1922891	7353351		3							

Reference Statistics:															
Total Instruction References			197365478												
Data Reads			60413211												
Data writes			25986851												
Total Data References			86400062												
Total References			283765540												
Miss Statistics:															
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)	
0	14733992	7.4653	17111983	28.3249	2467419	9.4949	19579402	22.6613	34313394	12.0922	9525760	24787510	124		
1	7955988	4.0311	9511514	15.7441	1506417	5.7968	11017931	12.7522	18973919	6.6865	5077360	13896307	252		
2	3629964	1.8392	3615836	5.9852	818052	3.1479	4433888	5.1318	8063852	2.8417	1855597	6207747	508		
3	1352554	0.6853	1923959	3.1847	259683	0.9993	2183642	2.5274	3536196	1.2462	860876	2674811	509		
4	6066645	3.0738	21068805	34.8745	5049067	19.4293	26117872	30.2290	32184517	11.3419	9652687	22531580	250		
5	5027332	2.5472	19994443	33.0961	4642566	17.8651	24637011	28.5150	29664343	10.4538	6278659	23385434	250		
6	3384002	1.7146	19299145	31.9452	4462279	17.1713	23761424	27.5016	27145426	9.5661	5986612	21158564	250		
7	5239231	2.6546	23847634	39.4742	3671018	14.1264	27518652	31.8503	32757883	11.5440	15122893	17634866	124		
8	4643111	2.3525	20191087	33.4216	3236386	12.4539	23427473	27.1151	28070584	9.8922	8438096	19632364	124		
9	2939946	1.4896	19254334	31.8711	3159903	12.1596	22414237	25.9424	25354183	8.9349	7204076	18149983	124		
10	5265285	2.6678	27647107	45.7633	3935017	15.1423	31582124	36.5534	36847409	12.9852	23487388	13359960	61		
11	4184315	2.1201	22136601	36.6420	2868716	11.0391	25005317	28.9413	29189632	10.2865	14092472	15097099	61		
12	3301257	1.6727	22629919	37.4586	2749314	10.5796	25379233	29.3741	28680490	10.1071	11941180	16739249	61		
13	3297438	1.6707	16767809	27.7552	3580919	13.7797	20348728	23.5518	23646166	8.3330	6212976	17432684	506		
14	2009299	1.0181	13487383	22.3252	3075514	11.8349	16562897	19.1700	18572196	6.5449	2942471	15629219	506		
15	1767644	0.8956	10949622	18.1245	2667886	10.2663	13617508	15.7610	15385152	5.4218	2778395	12606251	506		
16	2383453	1.2076	19885407	32.9157	2558116	9.8439	22443523	25.9763	24826976	8.7491	11105554	13721170	252		
17	1603663	0.8125	13387377	22.1597	2255217	8.6783	15642594	18.1048	17246257	6.0776	3283133	13962872	252		
18	1565303	0.7931	11752256	19.4531	1905488	7.3325	13657744	15.8076	15223047	5.3647	3659777	1156018	252		
19	1976369	1.0014	20791098	34.4148	2648988	10.1936	23440086	27.1297	25416455	8.9569	13893950	11522380	125		
20	1309220	0.6633	16511427	27.3308	1666971	6.4147	18178398	21.0398	19487618	6.8675	7100955	12386538	125		
21	1298598	0.6580	15693384	25.9767	1507333	5.8004	17200717	19.9082	18499315	6.5192	6620544	11878646	125		
22	2585730	1.3101	10587061	17.5244	2446141	9.4130	13033202	15.0847	15618932	5.5042	3211826	12406090	1016		
23	826419	0.4187	7521837	12.4506	1654138	6.3653	9175975	10.6203	10002394	3.5249	1768930	8232446	1018		
24	566490	0.2870	3989905	6.6044	1026095	3.9485	5016000	5.8056	5582490	1.9673	1140075	4441397	1018		
25	1836614	0.9306	9761843	16.1585	1814493	6.9824	11576336	13.3985	13412950	4.7268	3309544	10102898	508		
26	747689	0.3788	7704352	12.7528	1245750	4.7938	8950102	10.3589	9697791	3.4175	2113454	7583829	508		
27	564687	0.2861	4194660	6.9433	791513	3.0458	4986173	5.7710	5550860	1.9561	1394163	4156189	508		
28	1529552	0.7750	10302378	17.0532	1334490	5.1353	11636868	13.4686	13166420	4.6399	4176193	8989974	253		
29	624136	0.3162	9957029	16.4815	807181	3.1061	10764210	12.4586	11388346	4.0133	4499917	6888176	253		
30	522451	0.2647	6878876	11.3864	446648	1.7187	7325524	8.4786	7847975	2.7657	2662190	5185532	253		
31	1452006	0.7357	6871424	11.3740	1235969	4.7561	8107393	9.3835	9559399	3.3688	2885035	6673357	1007		
32	351236	0.1780	3045084	5.0404	614984	2.3665	3660068	4.2362	4011304	1.4136	796646	2312639	1019		
33	82589	0.0418	1395305	2.3096	297155	1.1435	1692460	1.9589	1775049	0.6255	367079	1406950	1020		
34	1241556	0.6291	7733697	12.8013	918076	3.5328	8651773	10.0136	9893329	3.4864	3963011	5929810	508		
35	287963	0.1459	3116622	5.1588	412029	1.5855	3528651	4.0841	3816614	1.3450	914720	2901385	509		
36	105590	0.0535	1817606	3.0086	187240	0.7205	2004846	2.3204	2110436	0.7437	638498	1471429	509		
37	978247	0.4957	9773521	16.1778	799544	3.0767	10573065	12.2373	11551312	4.0707	6410831	5140228	253		
38	299892	0.1519	5876335	9.7269	351855	1.3540	6228190	7.2085	6528082	2.3005	3199290	3328539	253		
39	128968	0.0653	3212333	5.3173	141441	0.5443	3353774	3.8817	3482742	1.2273	1559631	1922858	253		

Table 20: Alvinn w/ Operating System, Operating System Data

Reference Statistics:															
Total Instruction References			5430587523												
Data Reads			1475426841												
Data writes			513415325												
Total Data References			1988842166												
Total References			7419429689												
Miss Statistics:															
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)	
0	26511909	0.4882	79579417	5.3937	3575502	0.6964	83154919	4.1811	109666828	1.4781					
1	14534965	0.2676	51882856	3.5165	2303584	0.4487	54186440	2.7245	68721405	0.9262					
2	5554089	0.1023	37693320	2.5547	1253818	0.2442	38947138	1.9583	44501227	0.5998					
3	2078295	0.0383	18062370	1.2242	459252	0.0895	18521622	0.9313	20599917	0.2776					
4	20945006	0.3857	167238878	11.3349	7234450	1.4091	174473328	8.7726	195418334	2.6339					
5	19541133	0.3598	143766881	9.7441	5805188	1.1307	149572069	7.5206	169113202	2.2793					
6	18227072	0.3356	136496461	9.2513	5569699	1.0848	142066160	7.1432	160293232	2.1605					
7	16411145	0.3022	141782833	9.6096	6101497	1.1884	147884330	7.4357	164295475	2.2144					
8	15605108	0.2874	93926726	6.3661	4472334	0.8711	98399060	4.9476	114004168	1.5366					
9	13677086	0.2519	86475402	5.8610	3993669	0.7779	90469071	4.5488	104146157	1.4037					
10	13754317	0.2533	159879374	10.8361	6729674	1.3108	166609048	8.3772	180363365	2.4310					
11	11837583	0.2180	77585396	5.2585	4069850	0.7927	81655246	4.1057	93492829	1.2601					
12	10573114	0.1947	72781469	4.9329	3806125	0.7413	76587594	3.8509	87160708	1.1748					
13	12977923	0.2390	126779647	8.5927	5081336	0.9897	131860983	6.6300	144838906	1.9522					
14	9869298	0.1817	110507950	7.4899	4019110	0.7828	114527060	5.7585	124396358	1.6766					
15	6262391	0.1153	119152275	8.0758	3554444	0.6923	122706719	6.1698	128969110	1.7383					
16	9618168	0.1771	100152029	6.7880	4243604	0.8265	104395633	5.2491	114013801	1.5367					
17	8351822	0.1538	68767130	4.6608	3067314	0.5974	71834444	3.6119	80186266	1.0808					
18	7441601	0.1370	71873445	4.8714	2564898	0.4996	74438343	3.7428	81879944	1.1036					
19	7538461	0.1388	101849456	6.9031	4407431	0.8585	106256887	5.3427	113795348	1.5337					
20	6495502	0.1196	52180858	3.5367	2353761	0.4585	54534619	2.7420	61030121	0.8226					
21	6318534	0.1164	53084257	3.5979	2089049	0.4069	55173306	2.7741	61491840	0.8288					
22	9280794	0.1709	95696484	6.4860	3712743	0.7231	99409227	4.9983	108690021	1.4649					
23	4127288	0.0760	74931957	5.0787	2300700	0.4481	77232657	3.8833	81359945	1.0966					
24	1636352	0.0301	67631637	4.5839	1438839	0.2802	69070476	3.4729	70706828	0.9530					
25	7052310	0.1299	63671624	4.3155	3230638	0.6292	66902262	3.3639	73954572	0.9968					
26	4009543	0.0738	44092090	2.9884	1852897	0.3609	45944987	2.3101	49954530	0.6733					
27	1611699	0.0297	38212045	2.5899	1162800	0.2265	39374845	1.9798	40986544	0.5524					
28	5515854	0.1016	55734939	3.7775	2903537	0.5655	58638476	2.9484	64154330	0.8647					
29	3166730	0.0583	31041898	2.1039	1304532	0.2541	32346430	1.6264	35513160	0.4787					
30	2231557	0.0411	26534704	1.7984	838745	0.1634	27373449	1.3764	29605006	0.3990					
31	3919933	0.0722	47314667	3.2068	2090363	0.4071	49405030	2.4841	53324963	0.7187					
32	888722	0.0164	35170946	2.3838	1026866	0.2000	36197812	1.8200	37086534	0.4999					
33	102769	0.0019	31734287	2.1509	470398	0.0916	32204685	1.6193	32307454	0.4354					
34	3293971	0.0607	36195673	2.4532	1729993	0.3370	37925666	1.9069	41219637	0.5556					
35	1071981	0.0197	20258142	1.3730	857440	0.1670	21115582	1.0617	22187563	0.2990					
36	245737	0.0045	17591105	1.1923	432770	0.0843	18023875	0.9062	18269612	0.2462					
37	2523243	0.0465	45594576	3.0903	1864371	0.3631	47458947	2.3863	49982190	0.6737					
38	1166952	0.0215	16243688	1.1009	809290	0.1576	17052978	0.8574	18219930	0.2456					
39	332482	0.0061	12025461	0.8150	401044	0.0781	12426505	0.6248	12758987	0.1720					

Table 21: Alvinn w/ Operating System, Combined Data

Table 22: Compress and GCC w/ Operating System, Compress Data

Reference Statistics:														
Total Instruction References		87045885												
Data Reads		22411994												
Data writes		8521651												
Total Data References		30933645												
Total References		117979530												
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	1498727	1.7218	5111048	22.8050	219067	2.5707	5330115	17.2308	6828842	5.7882	2990684	3401796	3359028	4
1	945225	1.0859	4617692	20.6037	141510	1.6606	4759202	15.3852	5704427	4.8351	947086	2751523	2266786	4
2	224332	0.2577	3703986	16.5268	71570	0.8399	3775556	12.2053	3999888	3.3903	782543	1841532	1652508	4
3	56335	0.0647	3355032	14.9698	39180	0.4598	3394212	10.9726	3450547	2.9247	653539	1869438	927566	4
4	2321467	2.6669	5476227	24.4344	282158	3.3111	5758385	18.6153	8079852	6.8485	971277	4858237	3791656	6
5	990778	1.1382	4717087	21.0472	184074	2.1601	4901161	15.8441	5891939	4.9940	1033675	2309277	2404433	6
6	829248	0.9527	4494945	20.0560	151323	1.7757	4646268	15.0201	5475516	4.6411	1030056	1853536	2339753	6
7	1531662	1.7596	5842933	26.0706	519792	6.0997	6362725	20.5689	7894387	6.6913	741802	5381938	4739872	4
8	673638	0.7739	5451247	24.3229	284766	3.3417	5736013	18.5430	6409651	5.4329	1355069	3448973	2245936	4
9	620694	0.7131	4785458	21.3522	214598	2.5183	5000056	16.1638	5620750	4.7642	2434743	2383567	2460653	4
10	1315094	1.5108	7985939	35.6324	606797	7.1207	8592736	27.7780	9907830	8.3979	523800	8125374	4198423	4
11	482512	0.5543	6511508	29.0537	401943	4.7167	6913451	22.3493	7395963	6.2689	2652859	5291458	4906594	4
12	450616	0.5177	5164297	23.0426	328713	3.8574	5493010	17.7574	5943626	5.0378	1197389	3714170	1621100	4
13	741824	0.8522	4910633	21.9107	202880	2.3808	5113513	16.5306	5855337	4.9630	940786	3044327	1870218	6
14	492359	0.5656	4102808	18.3063	118938	1.3957	4221746	13.6477	4714105	3.9957	884151	2176337	1970561	6
15	275731	0.3168	3959052	17.6649	91365	1.0722	4050417	13.0939	4326148	3.6669	823013	2104759	881733	6
16	501909	0.5766	5264529	23.4898	422086	4.9531	5686615	18.3833	6188524	5.2454	789453	3778741	2418292	4
17	392486	0.4509	4294334	19.1609	167562	1.9663	4461896	14.4241	4854382	4.1146	834351	2357218	2230475	4
18	266031	0.3056	4199640	18.7384	107409	1.2604	4307049	13.9235	4573080	3.8762	824533	2099500	1649043	4
19	356651	0.4097	6977312	31.1320	473195	5.5529	7450507	24.0854	7807158	6.6174	680895	5788349	694447	4
20	328810	0.3777	4977349	22.2084	248266	2.9134	5225615	16.8930	5554425	4.7080	714505	3285748	753481	4
21	252261	0.2898	4569806	20.3900	171217	2.0092	4741023	15.3264	4993284	4.2323	786706	2512091	1694483	4
22	268272	0.3082	3924079	17.5088	128609	1.5092	4052688	13.1012	4320960	3.6625	608108	2371899	1340946	7
23	190444	0.2188	3730004	16.6429	90613	1.0633	3820617	12.3510	4011061	3.3998	714080	2231979	1064996	6
24	60562	0.0696	3602878	16.0757	74627	0.8757	3677505	11.8884	3738067	3.1684	657301	2261915	818844	7
25	198900	0.2285	4076345	18.1882	134426	1.5775	4210771	13.6123	4409671	3.7377	596722	2560321	1252623	5
26	160089	0.1839	3878708	17.3064	102744	1.2057	3981452	12.8709	4141541	3.5104	705643	2317764	1118130	4
27	65057	0.0747	3715160	16.5767	56035	0.6576	3771195	12.1912	3836252	3.2516	675955	2259529	1035444	4
28	143016	0.1643	4325929	19.3018	182247	2.1386	4508176	14.5737	4651192	3.9424	582741	2888221	1180226	4
29	136950	0.1573	4097636	18.2832	148564	1.7434	4246200	13.7268	4383150	3.7152	693288	2480213	1107306	4
30	80953	0.0930	3929986	17.5352	75500	0.8860	4005486	12.9486	4086439	3.4637	723844	2228626	1424921	4
31	111829	0.1285	3607865	16.0979	74686	0.8764	3682551	11.9047	3794380	3.2161	506201	2399075	889099	5
32	37061	0.0426	3441580	15.3560	48013	0.5634	3489593	11.2809	3526654	2.9892	534499	2308126	1598864	5
33	16448	0.0189	3397851	15.1609	40936	0.4804	3438787	11.1167	3455235	2.9287	514763	2375915	564552	5
34	82342	0.0946	3753468	16.7476	91129	1.0694	3844597	12.4285	3926939	3.3285	502981	2544547	879407	4
35	29719	0.0341	3576087	15.9561	57466	0.6744	3633553	11.7463	3663272	3.1050	569025	2329191	1723267	4
36	17018	0.0196	3530748	15.7538	37072	0.4350	3567820	11.5338	3584838	3.0385	567282	2365119	1463365	4
37	56842	0.0653	3923724	17.5073	107499	1.2615	4031223	13.0318	4088065	3.4651	490762	2709049	888250	4
38	22294	0.0256	3700435	16.5110	65792	0.7721	3766227	12.1752	3788521	3.2112	598070	2309659	880788	4
39	16515	0.0190	3660335	16.3320	48826	0.5730	3709161	11.9907	3725676	3.1579	637518	2236267	851887	4

Table 23: Compress and GCC w/ Operating System, GCC Data

Reference Statistics:														
Total Instruction References			68021687											
Data Reads			21218807											
Data writes			8094452											
Total Data References			29313259											
Total References			97334946											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	3296083	4.8456	2459680	11.5920	723132	8.9337	3182812	10.8579	6478895	6.6563	1028483	2883072	2924672	0
1	2599743	3.8219	1787496	8.4241	497655	6.1481	2285151	7.7956	4884894	5.0186	1233408	2131364	1951283	0
2	1770928	2.6035	1138682	5.3664	302147	3.7328	1440829	4.9153	3211757	3.2997	626709	1245364	1197420	0
3	942943	1.3862	761075	3.5868	145878	1.8022	906953	3.0940	1849896	1.9005	370904	931808	547184	0
4	4122896	6.0611	3081905	14.5244	1145900	14.1566	4227805	14.4228	8350701	8.5793	1260398	4669629	4813698	0
5	4105985	6.0363	2607875	12.2904	985282	12.1723	3593157	12.2578	7699142	7.9099	1390621	1635722	3721478	0
6	4066501	5.9782	2510744	11.8326	941974	11.6373	3452718	11.7787	7519219	7.7251	1453780	2635897	3429542	0
7	3051967	4.4868	3263379	15.3797	949620	11.7317	4212999	14.3723	7264966	7.4639	3074513	4339113	4513198	0
8	3042002	4.4721	2603518	12.2699	758076	9.3654	3361594	11.4678	6403596	6.5789	1060566	2186502	3182425	0
9	3000475	4.4111	2441452	11.5061	711702	8.7925	3153154	10.7568	6153629	6.3221	1066864	2392483	2694282	0
10	2254331	3.3141	3580808	16.8756	842419	10.4074	4423227	15.0895	6677558	6.8604	722055	4551380	4684193	0
11	2227303	3.2744	2701492	12.7316	616988	7.6224	3318480	11.3207	5545783	5.6976	804760	1532627	3208396	0
12	2172123	3.1933	2343990	11.0468	539195	6.6613	2883185	9.8358	5055308	5.1937	869941	2362408	2548206	0
13	3019182	4.4386	2345948	11.0560	867221	10.7138	3213169	10.9615	6232351	6.4030	1072934	1888358	3271059	0
14	2782569	4.0907	1859600	8.7639	678210	8.3787	2537810	8.6575	5320379	5.4661	1117905	1663299	2539175	0
15	2748894	4.0412	1699195	8.0080	616873	7.6209	2316068	7.9011	5064962	5.2036	1136406	1903892	2531096	0
16	2303255	3.3861	2454529	11.5677	717515	8.8643	3172044	10.8212	5475299	5.6252	865327	2116430	2972214	0
17	2140311	3.1465	1966915	9.2697	555454	6.8622	2522369	8.6049	4662680	4.7903	919720	1681856	2061104	0
18	2145056	3.1535	1870759	8.8165	525646	6.4939	2396405	8.1752	4541461	4.6658	953056	1666212	1922193	0
19	1755107	2.5802	2683167	12.6452	627119	7.7475	3310286	11.2928	5065393	5.2041	2395163	2755438	2999482	0
20	1648949	2.4242	2012736	9.4856	448882	5.5456	2461618	8.3976	4110567	4.2231	1394966	1569750	1774823	0
21	1667577	2.4515	1920310	9.0500	422896	5.2245	2343206	7.9937	4010783	4.1206	1070674	1721568	1503880	0
22	2087836	3.0694	1569687	7.3976	561304	6.9344	2130991	7.2697	4218827	4.3343	834523	1348692	2035612	0
23	1641342	2.4130	1233972	5.8155	450730	5.5684	1684702	5.7472	3326044	3.4171	745690	1065662	1514692	0
24	1367769	2.0108	1033688	4.8716	376112	4.6465	1409800	4.8094	2777569	2.8536	611991	818260	1347318	0
25	1625495	2.3897	1673006	7.8845	462669	5.7159	2135675	7.2857	3761170	3.8642	683684	1261991	1815495	0
26	1322614	1.9444	1352598	6.3745	367772	4.5435	1720370	5.8689	3042984	3.1263	650922	1124145	1267917	0
27	1177414	1.7309	1190642	5.6113	313719	3.8757	1504361	5.1320	2681775	2.7552	587366	902114	1192295	0
28	1275752	1.8755	1848471	8.7115	403700	4.9874	2252171	7.6831	3527923	3.6245	600138	1607075	1741760	0
29	1080111	1.5879	1518438	7.1561	315306	3.8953	1833744	6.2557	2913855	2.9936	588300	1195514	1109104	0
30	1022734	1.5035	1406373	6.6280	289120	3.5718	1695493	5.7840	2718227	2.7927	580149	1145049	993029	0
31	943226	1.3867	1107279	5.2184	312804	3.8644	1420083	4.8445	2363309	2.4280	502693	893760	966856	0
32	712934	1.0481	799519	3.7680	211800	2.6166	1011319	3.4500	1724253	1.7715	351881	686665	685707	0
33	511020	0.7513	683195	3.2198	184557	2.2800	867752	2.9603	1378772	1.4165	279199	565539	534034	0
34	741491	1.0901	1232346	5.8078	269928	3.3347	1502274	5.1249	2243765	2.3052	446326	884330	913109	0
35	588476	0.8651	942106	4.4400	180063	2.2245	1122169	3.8282	1710645	1.7575	340286	771941	598418	0
36	457750	0.6729	844133	3.9782	159716	1.9732	1003849	3.4246	1461599	1.5016	296742	653623	511234	0
37	605436	0.8901	1386455	6.5341	276242	3.4127	1662697	5.6722	2268133	2.3302	407800	892105	968228	0
38	495018	0.7277	1106104	5.2128	177910	2.1979	1284014	4.3803	1779032	1.8277	352377	886270	540385	0
39	420627	0.6184	1074871	5.0657	169522	2.0943	1244393	4.2452	1665020	1.7106	327173	854522	483325	0

Table 24: Compress and GCC w/ Operating System, Operating System Data

Reference Statistics:														
Total Instruction References			28102411											
Data Reads			7468658											
Data writes			4160003											
Total Data References			11628661											
Total References			39731072											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	2346796	8.3509	1982824	26.5486	569333	13.6859	2552157	21.9471	4898953	12.3303	2945432	901293	1052104	124
1	1914805	6.8137	1550741	20.7633	467924	11.2482	2018665	17.3594	3933470	9.9002	2085936	919618	927664	252
2	1386288	4.9330	1082465	14.4934	345335	8.3013	1427800	12.2783	2814088	7.0828	1404586	770674	638320	508
3	785675	2.7958	774395	10.3686	202941	4.8784	977336	8.4045	1763011	4.4374	738329	649223	374951	508
4	3221708	11.4642	2194271	29.3797	829816	19.9475	3024087	26.0055	6245795	15.7202	4014003	945006	1286536	250
5	3163371	11.2566	1994223	26.7012	768434	18.4720	2762657	23.7573	5926028	14.9153	3501622	995605	1428551	250
6	3193637	11.3643	1940224	25.9782	760303	18.2765	2700527	23.2230	5894164	14.8351	3410221	986064	1497629	250
7	2279492	8.1114	2271720	30.4167	626733	15.0657	2898453	24.9251	5177945	13.0325	3475542	721143	981136	124
8	2252370	8.0149	1972668	26.4126	548318	13.1807	2520986	21.6791	4773356	12.0142	2879271	800072	1093889	124
9	2265310	8.0609	1904701	25.5026	533695	12.8292	2438396	20.9688	4703706	11.8389	2745405	844691	1113486	124
10	1673619	5.9554	2325787	31.1406	555447	13.3521	2881234	24.7770	4554853	11.4642	3308977	511151	734665	60
11	1659018	5.9035	1980406	26.5162	444320	10.6808	2424726	20.8513	4083744	10.2785	2693052	571880	818752	60
12	1662297	5.9151	1869147	25.0265	413527	9.9405	2282674	19.6297	3944971	9.9292	2466649	592292	885970	60
13	2646875	9.4187	1774284	23.7564	700216	16.8321	2474500	21.2793	5121375	12.8901	3107461	922617	1090791	506
14	2372854	8.4436	1562768	20.9243	622838	14.9721	2185606	18.7950	4558460	11.4733	2556197	874443	1127314	506
15	2192995	7.8036	1454613	19.4762	611808	14.7069	2066421	17.7701	4259416	10.7206	2299809	823907	1135194	506
16	1883037	6.7006	1885878	25.2506	520172	12.5041	2406050	20.6907	4289087	10.7953	2634215	772013	882607	252
17	1734161	6.1709	1572077	21.0490	453415	10.8994	2025492	17.4181	3759653	9.4628	2005494	815303	938604	252
18	1654535	5.8875	1519065	20.3392	442582	10.6390	1961647	16.8691	3616182	9.1016	1838515	807363	970052	252
19	1400935	4.9851	1866650	24.9931	454914	10.9354	2321564	19.9642	3722499	9.3692	2324478	669988	727909	124
20	1325322	4.7160	1623555	21.7382	357107	8.5843	1980662	17.0326	3305984	8.3209	1825454	698933	781473	124
21	1282233	4.5627	1526430	20.4378	340042	8.1741	1866472	16.0506	3148705	7.9250	1576634	759626	812321	124
22	1908098	6.7898	1375650	18.4190	596045	14.3280	1971695	16.9555	3879793	9.7651	2436738	600177	841861	1017
23	1695769	6.0342	1105547	14.8025	499117	11.9980	1604664	13.7992	3300433	8.3069	1840212	713266	745937	1018
24	1445729	5.1445	973920	13.0401	456646	10.9771	1430566	12.3021	2876295	7.2394	1606537	657761	610980	1017
25	1368708	4.8704	1418567	18.9936	431756	10.3787	1850323	15.9117	3219031	8.1020	1938411	587295	692818	507
26	1232775	4.3867	1166839	15.6231	361862	8.6986	1528701	13.1460	2761476	6.9504	1404694	699588	656686	508
27	1076492	3.8306	1083626	14.5090	337796	8.1201	1421422	12.2234	2497914	6.2871	1234078	674613	588715	508
28	1029580	3.6637	1461182	19.5642	349823	8.4092	1811005	15.5736	2840585	7.1495	1657458	576947	605928	252
29	936558	3.3327	1263986	16.9239	288057	6.9244	1552043	13.3467	2488601	6.2636	1206759	686488	595102	252
30	851957	3.0316	1209837	16.1989	276356	6.6432	1486193	12.7804	2338150	5.8849	1034005	712766	591127	252
31	943273	3.3566	1108793	14.8459	380555	9.1480	1489348	12.8076	2432621	6.1227	1422865	501546	507191	1019
32	777341	2.7661	790038	10.5780	282248	6.7848	1072286	9.2211	1849627	4.6554	962381	531863	354364	1019
33	694886	2.4727	664638	8.8990	245756	5.9076	910394	7.8289	1605280	4.0404	810440	513780	280041	1019
34	707908	2.5190	1163339	15.5763	306375	7.3648	1469714	12.6387	2177622	5.4809	1227909	498063	451142	508
35	590235	2.1003	884949	11.8488	225709	5.4257	1110658	9.5510	1700893	4.2810	791176	562142	347067	508
36	525479	1.8699	797538	10.6785	200547	4.8208	998085	8.5830	1523564	3.8347	659135	566095	297826	508
37	564227	2.0078	1277802	17.1089	261602	6.2885	1539404	13.2380	2103631	5.2947	1204886	486911	411582	252
38	463834	1.6505	1034358	13.8493	188460	4.5303	1222818	10.5156	1686652	4.2452	736025	592591	357754	252
39	414239	1.4740	970269	12.9912	173938	4.1812	1144207	9.8395	1558446	3.9225	593578	634887	329729	252

Table 25: Compress and GCC w/ Operating System, Combined Data

Reference Statistics:																			
Total Instruction References	183169983																		
Data Reads	51099459																		
Data writes	20776106																		
Total Data References	71875565																		
Total References	255045548																		
Miss Statistics:																			
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	Int(0)	Int(1)	Int(2)	Int(3)					
0	7141606	3.8989	9553552	18.6960	1511532	7.2753	11065084	15.3948	18206690	7.1386									
1	5459773	2.9807	7955929	15.5695	1107089	5.3287	9063018	12.6093	14522791	5.6942									
2	3381548	1.8461	5925133	11.5953	719052	3.4610	6644185	9.2440	10025733	3.9310									
3	1784953	0.9745	4890502	9.5706	387999	1.8675	5278501	7.3439	7063454	2.7695									
4	9666071	5.2771	10752403	21.0421	2257874	10.8676	13010277	18.1011	22676348	8.8911									
5	8260134	4.5095	9319185	18.2373	1937790	9.3270	11256975	15.6618	19517109	7.6524									
6	8089386	4.4163	8945913	17.5069	1853600	8.9218	10799513	15.0253	18888899	7.4061									
7	6863121	3.7469	11378032	22.2664	2096145	10.0892	13474177	18.7465	20337298	7.9740									
8	5968010	3.2582	10027433	19.6234	1591160	7.6586	11618593	16.1649	17586603	6.8955									
9	5886479	3.2137	9131611	17.8703	1459995	7.0273	10591606	14.7360	16478085	6.4608									
10	5243044	2.8624	13892534	27.1872	2004663	9.6489	15897197	22.1177	21140241	8.2888									
11	4366833	2.3851	11193406	21.9051	1463251	7.0430	12656657	17.6091	17025490	6.6755									
12	4285036	2.3394	9377434	18.3513	1281435	6.1678	10658869	14.8296	14943905	5.8593									
13	6407881	3.4983	9030865	17.6731	1770317	8.5209	10801182	15.0276	17209063	6.7474									
14	5647782	3.0834	7525176	14.7265	1419986	6.8347	8945162	12.4453	14592944	5.7217									
15	5217620	2.8485	7112860	13.9196	1320046	6.3537	8432906	11.7326	13650526	5.3522									
16	4688201	2.5595	9604936	18.7966	1659773	7.9889	11264709	15.6725	15952910	6.2549									
17	4266958	2.3295	7833326	15.3296	1176431	5.6624	9009757	12.5352	13276715	5.2056									
18	4065622	2.2196	7589464	14.8523	1075637	5.1773	8665101	12.0557	12730723	4.9915									
19	3512693	1.9177	11527129	22.5582	1555228	7.4857	13082357	18.2014	16595050	6.5067									
20	3303081	1.8033	8613640	16.8566	1054255	5.0744	9667895	13.4509	12970976	5.0857									
21	3202071	1.7481	8016546	15.6881	934155	4.4963	8950701	12.4531	12152772	4.7649									
22	4264206	2.3280	6869416	13.4432	1285958	6.1896	8155374	11.3465	12419580	4.8696									
23	3527555	1.9258	6069523	11.8779	1040460	5.0080	7109983	9.8921	10637538	4.1708									
24	2874060	1.5691	5610486	10.9795	907385	4.3674	6517871	9.0683	9391931	3.6825									
25	3193103	1.7432	7167918	14.0274	1028851	4.9521	8196769	11.4041	11389872	4.4658									
26	2715478	1.4825	6398145	12.5210	832378	4.0064	7230523	10.0598	9946001	3.8997									
27	2318963	1.2660	5989428	11.7211	707550	3.4056	6696978	9.3175	9015941	3.5350									
28	2448348	1.3367	7635582	14.9426	935770	4.5041	8571352	11.9253	11019700	4.3207									
29	2153619	1.1757	6880060	13.4641	751927	3.6192	7631987	10.6183	9785606	3.8368									
30	1955644	1.0677	6546196	12.8107	640976	3.0852	7187172	9.9995	9142816	3.5848									
31	1998328	1.0910	5823937	11.3973	768045	3.6968	6591982	9.1714	8590310	3.3681									
32	1527336	0.8338	5031137	9.8458	542061	2.6091	5573198	7.7540	7100534	2.7840									
33	1222354	0.6673	4745684	9.2872	471249	2.2682	5216933	7.2583	6439287	2.5248									
34	1531741	0.8362	6149153	12.0337	667432	3.2125	6816585	9.4839	8348326	3.2733									
35	1208430	0.6597	5403142	10.5738	463238	2.2297	5866380	8.1619	7074810	2.7739									
36	1000247	0.5461	5172419	10.1223	397335	1.9125	5569754	7.7492	6570001	2.5760									
37	1226505	0.6696	6587981	12.8925	645343	3.1062	7233324	10.0637	8459829	3.3170									
38	981146	0.5356	5840897	11.4304	432162	2.0801	6273059	8.7277	7254205	2.8443									
39	851381	0.4648	5705475	11.1654	392286	1.8882	6097761	8.4838	6949142	2.7247									

Table 26: Compress and Espresso w/ Operating System, Compress Data

Reference Statistics:														
Total Instruction References		87045885												
Data Reads		22411994												
Data writes		8521651												
Total Data References		30933645												
Total References		117979530												
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	1249023	1.4349	4875818	21.7554	172163	2.0203	5047981	16.3187	6297004	5.3374	883351	4034776	1378873	4
1	741902	0.8523	4377883	19.5337	98848	1.1600	4476731	14.4720	5216633	4.4233	838125	3380432	1000072	4
2	117209	0.1347	3518959	15.7012	48755	0.5721	3567714	11.5334	3684923	3.1234	628861	2333977	722081	4
3	16120	0.0185	3189503	14.2312	26317	0.3088	3215820	10.3959	3231940	2.7394	514123	2296081	421732	4
4	1849816	2.1251	5340555	23.8290	256805	3.0136	5597360	18.0947	7447176	6.3123	783065	5098571	4512527	6
5	401699	0.4615	4548709	20.2959	153069	1.7962	4701778	15.1996	5103477	4.3257	843730	2632417	2446822	6
6	201315	0.2313	4261903	19.0162	114732	1.3464	4376635	14.1485	4577950	3.8803	773060	2253301	2253172	6
7	1250653	1.4368	5737477	25.6000	496822	5.8301	6234299	20.1538	7484952	6.3443	2877211	5651216	5448885	4
8	289833	0.3330	5318351	23.7299	250565	2.9403	5568916	18.0028	5858749	4.9659	2525379	3792984	1311480	4
9	172813	0.1985	4592674	20.4920	175143	2.0553	4767817	15.4130	4940630	4.1877	733622	2800032	2545557	4
10	1174475	1.3493	7923558	35.3541	588258	6.9031	8511816	27.5164	9686291	8.2101	6822276	8333952	6102541	4
11	274821	0.3157	6436516	28.7191	377823	4.4337	6814339	22.0289	7089160	6.0088	3228447	5515239	3284533	4
12	182258	0.2094	5079384	22.6637	308791	3.6236	5388175	17.4185	5570433	4.7215	571101	3946244	3658545	4
13	287388	0.3302	4774522	21.3034	178018	2.0890	4952540	16.0102	5239928	4.4414	709318	3226647	1303957	6
14	180081	0.2069	3964947	17.6912	99890	1.1722	4064837	13.1405	4244918	3.5980	651288	2384590	1209034	6
15	44307	0.0509	3834975	17.1113	79069	0.9279	3914044	12.6530	3958351	3.3551	610058	2305047	1043240	6
16	206954	0.2378	5140078	22.9345	399427	4.6872	5539505	17.9077	5746459	4.8707	656058	4073328	3150908	4
17	156957	0.1803	4149660	18.5154	136988	1.6075	4286648	13.8576	4443605	3.7664	2583623	2725389	1044366	4
18	48353	0.0555	4029441	17.9789	74146	0.8701	4103587	13.2658	4151940	3.5192	656982	2518396	976558	4
19	178766	0.2054	6878721	30.6921	451106	5.2936	7329827	23.6953	7508593	6.3643	617205	6083762	3814355	4
20	156924	0.1803	4835808	21.5769	212715	2.4962	5048523	16.3205	5205447	4.4122	623073	3661100	921270	4
21	56163	0.0645	4397479	19.6211	127933	1.5013	4525412	14.6294	4581575	3.8834	680949	2961225	990581	4
22	61813	0.0710	3843685	17.1501	111179	1.3047	3954864	12.7850	4016677	3.4046	448678	2525508	1042485	6
23	41613	0.0478	3627375	16.1850	81617	0.9578	3708992	11.9902	3750605	3.1790	477003	2332664	2168708	6
24	11933	0.0137	3543961	15.8128	71530	0.8394	3615491	11.6879	3627424	3.0746	449532	2353971	823915	6
25	54146	0.0622	3973988	17.7315	105314	1.2358	4079302	13.1873	4133448	3.5035	469757	2837914	1808941	4
26	29388	0.0338	3749168	16.7284	81168	0.9525	3830336	12.3824	3859724	3.2715	525276	2585590	748854	4
27	13744	0.0158	3627558	16.1858	43176	0.5067	3670734	11.8665	3684478	3.1230	513538	2510736	660200	4
28	32859	0.0377	4198818	18.7347	147222	1.7276	4346040	14.0496	4378899	3.7116	494696	3219192	665007	4
29	26069	0.0299	3937632	17.5693	114428	1.3428	4052060	13.0992	4078129	3.4566	563142	2843748	671235	4
30	16373	0.0188	3782336	16.8764	44857	0.5264	3827193	12.3723	3843566	3.2578	594011	2623105	626446	4
31	41623	0.0478	3553160	15.8538	71584	0.8400	3624744	11.7178	3666367	3.1076	387856	2636434	642073	4
32	5874	0.0067	3371037	15.0412	41371	0.4855	3412408	11.0314	3418282	2.8974	390488	2488808	538981	5
33	3714	0.0043	3334909	14.8800	36893	0.4329	3371802	10.9001	3375516	2.8611	361123	2518790	495599	4
34	22731	0.0261	3679179	16.4161	80412	0.9436	3759591	12.1537	3782322	3.2059	416368	2834225	531725	4
35	3963	0.0046	3475737	15.5084	33515	0.3933	3509252	11.3445	3513215	2.9778	445681	2613314	454216	4
36	3011	0.0035	3433628	15.3205	24092	0.2827	3457720	11.1779	3460731	2.9333	441366	2619103	400258	4
37	20201	0.0232	3833700	17.1056	93217	1.0939	3926917	12.6946	3947118	3.3456	427638	3029862	489614	4
38	3082	0.0035	3589189	16.0146	39686	0.4657	3628875	11.7312	3631957	3.0785	500643	2683794	1597469	4
39	3210	0.0037	3534656	15.7713	25632	0.3008	3560288	11.5094	3563498	3.0204	533372	2618974	411148	4

Table 27: Compress and Espresso w/ Operating System, Espresso Data

Reference Statistics:																	
Total Instruction References		99475944															
Data Reads		24280822															
Data writes		4659787															
Total Data References		28940609															
Total References		128416553															
Miss Statistics:																	
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)			
0	935344	0.9403	1633589	6.7279	151318	3.2473	1784907	6.1675	2720251	2.1183	407172	1372386	940693	0			
1	693430	0.6971	1235501	5.0884	108165	2.3212	1343666	4.6428	2037096	1.5863	323283	997695	716118	0			
2	346611	0.3484	774431	3.1895	83916	1.8009	858347	2.9659	1204958	0.9383	209219	721268	274471	0			
3	194955	0.1960	413495	1.7030	40145	0.8615	453640	1.5675	648595	0.5051	103260	422215	123120	0			
4	1754647	1.7639	2803359	11.5456	237876	5.1049	3041235	10.5085	4795882	3.7346	1720322	1604938	2744011	0			
5	723285	0.7271	2254691	9.2859	210905	4.5261	2465596	8.5195	3188881	2.4832	565472	1619510	1003899	0			
6	690907	0.6945	2193038	9.0320	205265	4.4050	2398303	8.2870	3089210	2.4056	601744	1587510	939389	0			
7	1276331	1.2831	2506861	10.3244	197672	4.2421	2704533	9.3451	3980864	3.1000	350139	1903977	2450787	0			
8	512459	0.5152	1808852	7.4497	162506	3.4874	1971358	6.8117	2463817	1.9342	399146	1324436	760235	0			
9	479060	0.4816	1685322	6.9410	154974	3.3258	1840296	6.3589	2319356	1.8061	403651	1312312	513952	0			
10	1059126	1.0647	2651099	10.9185	184055	3.9499	2835154	9.7965	3894280	3.0325	242949	2741426	2802143	0			
11	403647	0.4058	1594340	6.5663	130100	2.7920	1724440	5.9585	2128087	1.6572	827396	921486	836099	0			
12	371854	0.3738	1405400	5.7881	117740	2.5267	1523140	5.2630	1894994	1.4757	331357	594073	515091	0			
13	1442306	1.4499	2261220	9.3128	182263	3.9114	2443483	8.4431	3885789	3.0259	376369	1298958	2210462	0			
14	430209	0.4325	1780515	7.3330	161543	3.4667	1942058	6.7105	2372267	1.8473	444211	1203443	724613	0			
15	342218	0.3440	1690616	6.9628	152118	3.2645	1842734	6.3673	2184952	1.7015	428014	1042176	714762	0			
16	1064894	1.0705	1911840	7.8739	149935	3.2176	2061775	7.1242	3126669	2.4348	266713	1012400	1847556	0			
17	311352	0.3130	1364899	5.6213	128020	2.7473	1492919	5.1586	1804271	1.4050	335612	1040354	428305	0			
18	252335	0.2537	1295229	5.3344	120189	2.5793	1415418	4.8908	1667753	1.2987	341173	972229	354351	0			
19	902485	0.9072	1915923	7.8907	133351	2.8617	2049274	7.0810	2951759	2.2986	202617	803868	1945274	0			
20	264215	0.2656	1163319	4.7911	103842	2.2285	1267161	4.3785	1531376	1.1925	257615	512589	355838	0			
21	209903	0.2110	1096330	4.5152	95221	2.0435	1191551	4.1172	1401454	1.0913	264634	936310	200510	0			
22	329169	0.3309	1706797	7.0294	150441	3.2285	1857238	6.4174	2186407	1.7026	264653	1039342	882412	0			
23	161030	0.1619	1338970	5.5145	116238	2.4945	1455208	5.0283	1616238	1.2586	288418	937609	390219	0			
24	118626	0.1193	1274638	5.2496	106320	2.2816	1380958	4.7717	1499584	1.1677	230875	391556	446659	0			
25	243207	0.2445	1403438	5.7800	124172	2.6648	1527610	5.2784	1770817	1.3790	206902	823217	740698	0			
26	115652	0.1163	997555	4.1084	91465	1.9629	1089020	3.7629	1204672	0.9381	225416	747453	231803	0			
27	92924	0.0934	904006	3.7231	83013	1.7815	987019	3.4105	1079943	0.8410	191780	660343	227820	0			
28	179854	0.1808	1319378	5.4338	108152	2.3210	1427530	4.9326	1607384	1.2517	169482	662757	775145	0			
29	98965	0.0995	840552	3.4618	74631	1.6016	915183	3.1623	1014148	0.7897	184754	669141	160253	0			
30	86783	0.0872	773758	3.1867	69738	1.4966	843496	2.9146	930279	0.7244	180267	207881	124833	0			
31	123580	0.1242	938463	3.8650	86705	1.8607	1025168	3.5423	1148748	0.8945	122672	641568	384508	0			
32	50236	0.0505	689103	2.8381	67758	1.4541	756861	2.6152	807097	0.6285	117753	538819	150525	0			
33	20035	0.0201	616656	2.5397	54932	1.1789	671588	2.3206	691623	0.5386	95049	495201	101373	0			
34	90085	0.0906	829877	3.4178	74155	1.5914	904032	3.1237	994117	0.7741	101030	531059	362028	0			
35	37653	0.0379	566669	2.3338	54576	1.1712	621245	2.1466	658898	0.5131	104103	454366	100429	0			
36	19013	0.0191	489106	2.0144	44868	0.9629	533974	1.8451	552987	0.4306	87246	400702	65039	0			
37	91309	0.0918	848865	3.4960	75427	1.6187	924292	3.1938	1015601	0.7909	92579	488952	434070	0			
38	34787	0.0350	566280	2.3322	48662	1.0443	614942	2.1248	649729	0.5060	100907	447535	101287	0			
39	20748	0.0209	486904	2.0053	41371	0.8878	528275	1.8254	549023	0.4275	87768	412080	49175	0			

Table 28: Compress and Espresso w/ Operating System, Operating System Data

Reference Statistics:																							
Total Instruction References		15541809																					
Data Reads		4310868																					
Data writes		2247254																					
Total Data References		6558122																					
Total References		22099931																					
Miss Statistics:																							
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)									
0	1255752	8.0798	1247680	28.9427	293096	13.0424	1540776	23.4942	2796528	12.6540	1505937	899817	400650	124									
1	963360	6.1985	930229	21.5787	249606	11.1072	1179835	17.9904	2143195	9.6977	981598	840476	320869	252									
2	634857	4.0848	624586	14.4886	204084	9.0815	828670	12.6358	1463527	6.6223	625068	629606	208345	508									
3	372165	2.3946	451765	10.4797	126377	5.6236	578142	8.8157	950307	4.3000	332581	513546	103672	508									
4	1410954	9.0784	1401229	32.5046	451974	20.1123	1853203	28.2581	3264157	14.7700	1988801	788811	486295	250									
5	1422455	9.1524	1309809	30.3839	423814	18.8592	1733623	26.4347	3156078	14.2809	1746702	851520	557606	250									
6	1364589	8.7801	1274236	29.5587	413885	18.4174	1688121	25.7409	3052710	13.8132	1677742	776542	598176	250									
7	1014947	6.5304	1476771	34.2569	329167	14.6475	1805938	27.5374	2820885	12.7642	1821502	653780	345479	124									
8	1029123	6.6216	1302201	30.2074	289981	12.9038	1592182	24.2780	2621305	11.8611	1487803	741307	392071	124									
9	1014544	6.5278	1264011	29.3215	281940	12.5460	1545951	23.5731	2560495	11.5860	1423159	738825	398387	124									
10	767231	4.9366	1545298	35.8466	314726	14.0049	1860024	28.3621	2627255	11.8881	1884252	503157	239786	60									
11	809662	5.2096	1310684	30.4042	246726	10.9790	1557410	23.7478	2367072	10.7108	1514959	569723	282330	60									
12	812093	5.2252	1246720	28.9204	230432	10.2539	1477152	22.5240	2289245	10.3586	1386773	575651	326761	60									
13	1050419	6.7587	1125472	26.1078	398219	17.7202	1523691	23.2336	2574110	11.6476	1488110	714270	371224	506									
14	906107	5.8301	999043	23.1750	364960	16.2403	1364003	20.7987	2270110	10.2720	1174275	656848	438481	506									
15	730511	4.7003	951138	22.0637	357857	15.9242	1308995	19.9599	2039506	9.2286	1001098	611099	426803	506									
16	759949	4.8897	1225091	28.4187	282965	12.5916	1508056	22.9952	2268005	10.2625	1345082	660719	261952	252									
17	684694	4.4055	1018630	23.6293	247897	11.0311	1266527	19.3123	1951221	8.8291	941615	677846	331508	252									
18	602086	3.8740	1011517	23.4643	244103	10.8623	1255620	19.1460	1857706	8.4059	859407	661299	336748	252									
19	581600	3.7422	1227888	28.4835	264246	11.7586	1492134	22.7525	2073734	9.3834	1253836	620962	198812	124									
20	537725	3.4599	1071009	24.8444	200980	8.9434	1271989	19.3956	1809714	8.1886	928955	626423	254212	124									
21	501997	3.2300	1030932	23.9147	193601	8.6150	1224533	18.6720	1726530	7.8124	780889	684033	261484	124									
22	643425	4.1400	863534	20.0316	353470	15.7290	1217004	18.5572	1860429	8.4183	1146498	451615	261298	1018									
23	543946	3.4999	710948	16.4920	309602	13.7769	1020550	15.5616	1564496	7.0792	798438	480163	284877	1018									
24	429623	2.7643	625706	14.5146	290141	12.9109	915847	13.9651	1345470	6.0881	664359	451282	228811	1018									
25	458033	2.9471	900375	20.8862	246372	10.9632	1146747	17.4859	1604780	7.2615	927815	472231	204226	508									
26	399612	2.5712	755001	17.5139	213906	9.5186	968907	14.7742	1368519	6.1924	617492	526628	223891	508									
27	322385	2.0743	699518	16.2268	203651	9.0622	903169	13.7718	1225554	5.5455	519874	513357	191815	508									
28	344183	2.2146	941125	21.8315	203833	9.0703	1144958	17.4586	1489141	6.7382	824801	496926	167162	252									
29	304383	1.9585	839558	19.4754	173995	7.7426	1013553	15.4549	1317936	5.9635	569871	565227	182586	252									
30	268701	1.7289	807514	18.7321	165699	7.3734	973213	14.8398	1241914	5.6195	467462	595270	178930	252									
31	279072	1.7956	679361	15.7593	223279	9.9356	902640	13.7637	1181712	5.3471	670621	388089	121982	1020									
32	214631	1.3810	501698	11.6380	180337	8.0248	682035	10.3999	896666	4.0573	387812	390425	117410	1019									
33	180799	1.1633	428697	9.9446	159781	7.1101	588478	8.9733	769277	3.4809	312475	361313	94469	1020									
34	205143	1.3199	739903	17.1637	181623	8.0820	921526	14.0517	1126669	5.0981	608961	416937	100263	508									
35	161124	1.0367	571147	13.2490	147141	6.5476	718288	10.9526	879412	3.9793	329289	445460	104155	508									
36	140927	0.9068	524760	12.1730	132024	5.8749	656784	10.0148	797711	3.6096	268748	440860	87595	508									
37	168186	1.0822	836537	19.4053	140753	6.2633	977290	14.9020	1145476	5.1832	625095	428273	91856	252									
38	130049	0.8368	691857	16.0491	112996	5.0282	804853	12.2726	934902	4.2303	333171	500608	100871	252									
39	116437	0.7492	662684	15.3724	103300	4.5967	765984	11.6799	882421	3.9929	261097	532430	88642	252									

Table 29: Compress and Espresso w/ Operating System, Combined Data

Reference Statistics:														
Total Instruction References			202063638											
Data Reads			51003684											
Data writes			15428692											
Total Data References			66432376											
Total References			268496014											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	3440119	1.7025	7757087	15.2089	616577	3.9963	8373664	12.6048	11813783	4.4000				
1	2398692	1.1871	6543613	12.8297	456619	2.9595	7000232	10.5374	9398924	3.5006				
2	1098677	0.5437	4917976	9.6424	336755	2.1827	5254731	7.9099	6353408	2.3663				
3	583240	0.2886	4054763	7.9499	192839	1.2499	4247602	6.3939	4830842	1.7992				
4	5015417	2.4821	9545143	18.7146	946655	6.1357	10491798	15.7932	15507215	5.7756				
5	2547439	1.2607	8113209	15.9071	787788	5.1060	8900997	13.3986	11448436	4.2639				
6	2256811	1.1169	7729177	15.1542	733882	4.7566	8463059	12.7394	10719870	3.9926				
7	3541931	1.7529	9721109	19.0596	1023661	6.6348	10744770	16.1740	14286701	5.3210				
8	1831415	0.9064	8429404	16.5270	703052	4.5568	9132456	13.7470	10963871	4.0834				
9	1666417	0.8247	7542007	14.7872	612057	3.9670	8154064	12.2742	9820481	3.6576				
10	3000832	1.4851	12119955	23.7629	1087039	7.0456	13206994	19.8804	16207826	6.0365				
11	1488130	0.7365	9341540	18.3154	754649	4.8912	10096189	15.1977	11584319	4.3145				
12	1366205	0.6761	7731504	15.1587	656963	4.2581	8388467	12.6271	9754672	3.6331				
13	2780113	1.3759	8161214	16.0012	758500	4.9162	8919714	13.4268	11699827	4.3575				
14	1516397	0.7505	6744505	13.2236	626393	4.0599	7370898	11.0953	8887295	3.3100				
15	1117036	0.5528	6476729	12.6986	589044	3.8178	7065773	10.6360	8182809	3.0476				
16	2031797	1.0055	8277009	16.2283	832327	5.3947	9109336	13.7122	11141133	4.1495				
17	1153003	0.5706	6533189	12.8092	512905	3.3244	7046094	10.6064	8199097	3.0537				
18	902774	0.4468	6336187	12.4230	438438	2.8417	6774625	10.1978	7677399	2.8594				
19	1662851	0.8229	10022532	19.6506	848703	5.5008	10871235	16.3644	12534086	4.6683				
20	958864	0.4745	7070136	13.8620	517537	3.3544	7587673	11.4216	8546537	3.1831				
21	768063	0.3801	6524741	12.7927	416755	2.7012	6941496	10.4490	7709559	2.8714				
22	1034407	0.5119	6414016	12.5756	615090	3.9867	7029106	10.5808	8063513	3.0032				
23	746589	0.3695	5677293	11.1311	507457	3.2890	6184750	9.3098	6931339	2.5815				
24	560182	0.2772	5444305	10.6743	467991	3.0333	5912296	8.8997	6472478	2.4106				
25	755386	0.3738	6277801	12.3085	475858	3.0842	6753659	10.1662	7509045	2.7967				
26	544652	0.2695	5501724	10.7869	386539	2.5053	5888263	8.8635	6432915	2.3959				
27	429053	0.2123	5231082	10.2563	329840	2.1378	5560922	8.3708	5989975	2.2309				
28	556896	0.2756	6459321	12.6644	459207	2.9763	6918528	10.4144	7475424	2.7842				
29	429417	0.2125	5617742	11.0144	363054	2.3531	5980796	9.0028	6410213	2.3875				
30	371857	0.1840	5363608	10.5161	280294	1.8167	5643902	8.4957	6015759	2.2405				
31	444275	0.2199	5170984	10.1385	381568	2.4731	5552552	8.3582	5996827	2.2335				
32	270741	0.1340	4561838	8.9441	289466	1.8762	4851304	7.3026	5122045	1.9077				
33	204548	0.1012	4380262	8.5881	251606	1.6308	4631868	6.9723	4836416	1.8013				
34	317959	0.1574	5248959	10.2913	336190	2.1790	5585149	8.4073	5903108	2.1986				
35	202740	0.1003	4613553	9.0455	235232	1.5246	4848785	7.2988	5051525	1.8814				
36	162951	0.0806	4447494	8.7199	200984	1.3027	4648478	6.9973	4811429	1.7920				
37	279696	0.1384	5519102	10.8210	309397	2.0053	5828499	8.7736	6108195	2.2750				
38	167918	0.0831	4847326	9.5039	201344	1.3050	5048670	7.5997	5216588	1.9429				
39	140395	0.0695	4684244	9.1841	170303	1.1038	4854547	7.3075	4994942	1.8603				

Reference Statistics:														
Total Instruction References			160240175											
Data Reads			50197333											
Data writes			19074845											
Total Data References			69272178											
Total References			229512353											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	Int(0)	Int(1)	Int(2)	Int(3)
0	7022905	4.3827	5032668	10.0258	1551453	8.1335	6584121	9.5047	13607026	5.9287	2511865	7408525	2533612	3
1	5166542	3.2242	3475694	6.9241	1037170	5.4374	4512864	6.5147	9679406	4.2174	2175838	4910549	2287801	3
2	3190279	1.9909	2074680	4.1330	576541	3.0225	2651221	3.8273	5841500	2.5452	2737510	2969652	1464869	3
3	1433315	0.8945	1159066	2.3090	207366	1.0871	1366432	1.9726	2799747	1.2199	825411	1331072	643261	3
4	9475631	5.9134	6473317	12.8957	2491250	13.0604	8964567	12.9411	18440198	8.0345	2877338	11072752	9371004	4
5	9277901	5.7900	5229038	10.4170	2039630	10.6928	7268668	10.4929	16546569	7.2094	5641373	8617694	7231324	4
6	9190519	5.7355	4844234	9.6504	1891660	9.9170	6735894	9.7238	15926413	6.9392	3155554	7966591	7552231	4
7	6958037	4.3423	6867432	13.6809	2083536	10.9230	8950968	12.9214	15909005	6.9317	2265315	10503766	10321709	3
8	6829794	4.2622	5202171	10.3634	1558541	8.1707	6760712	9.7596	13590506	5.9215	2624591	7506007	3601502	3
9	6733382	4.2021	4761685	9.4859	1422134	7.4555	6183819	8.9268	12917201	5.6281	2553600	6439078	6253971	3
10	5136311	3.2054	7724428	15.3881	1909611	10.0111	9634039	13.9075	14770350	6.4355	1744863	10786685	9262897	3
11	5011960	3.1278	5526317	11.0092	1304251	6.8375	6830568	9.8605	11842528	5.1599	4424804	7317319	6443208	3
12	4891843	3.0528	4765139	9.4928	1122489	5.8847	5887628	8.4993	10779471	4.6967	1961456	5869199	5641268	3
13	6960736	4.3439	4661066	9.2855	1826651	9.5762	6487717	9.3655	13448453	5.8596	2433960	7542904	3471585	4
14	6496350	4.0541	3458483	6.8898	1334065	6.9938	4792548	6.9184	11288898	4.9186	2315589	5825460	3147845	4
15	6312451	3.9394	3040529	6.0572	1162041	6.0920	4202570	6.0668	10515021	4.5815	2209900	5572267	3955965	4
16	5274315	3.2915	4859655	9.6811	1516497	7.9502	6376152	9.2045	11650467	5.0762	2066456	7022047	6413363	3
17	4972138	3.1029	3510089	6.9926	1022861	5.3624	4532950	6.5437	9505088	4.1414	2529407	4868903	2529294	3
18	4915984	3.0679	3156951	6.2891	895204	4.6931	4052155	5.8496	8968139	3.9075	2113825	4526839	3310356	3
19	4036125	2.5188	5426495	10.8103	1358867	7.1239	6785362	9.7952	10821487	4.7150	6460159	7052438	5010703	3
20	3834665	2.3931	3651682	7.2747	834165	4.3731	4485847	6.4757	8320512	3.6253	1874326	4302716	3131335	3
21	3822074	2.3852	3341470	6.6567	729425	3.8240	4070895	5.8767	7892969	3.4390	1988981	3730790	2173195	3
22	4878942	3.0448	2980159	5.9369	1155613	6.0583	4135772	5.9703	9014714	3.9278	1730613	4722418	4231672	4
23	3754955	2.3433	2109931	4.2033	856346	4.4894	2966277	4.2821	6721232	2.9285	1434504	3437830	2642828	4
24	3191678	1.9918	1721462	3.4294	701461	3.6774	2422923	3.4977	5614601	2.4463	1172257	3062532	1379808	4
25	3769986	2.3527	3070222	6.1163	916648	4.8055	3986870	5.7554	7756856	3.3797	1487718	4346255	1922880	3
26	3011260	1.8792	2132730	4.2487	625881	3.2812	2758611	3.9823	5769871	2.5140	1360430	2936394	1473044	3
27	2725094	1.7006	1757459	3.5011	492300	2.5809	2249759	3.2477	4974853	2.1676	1210089	2663727	1101034	3
28	2977461	1.8581	3364083	6.7017	779589	4.0870	4143672	5.9817	7121133	3.1027	1401971	4192005	1527154	3
29	2495651	1.5574	2339369	4.6603	506137	2.6534	2845506	4.1077	5341157	2.3272	1703742	2678091	2482710	3
30	2378836	1.4845	1965331	3.9152	405402	2.1253	2370733	3.4223	4749569	2.0694	1356440	2358083	1945071	3
31	2133385	1.3314	1841550	3.6686	518709	2.7193	2360259	3.4072	4493644	1.9579	1063946	2233732	1195963	3
32	1599303	0.9981	1122347	2.2359	310135	1.6259	1432482	2.0679	3031785	1.3210	730972	1545309	755500	4
33	1123693	0.7013	868273	1.7297	260236	1.3643	1128509	1.6291	2252202	0.9813	528335	1217204	506660	3
34	1709892	1.0671	1972644	3.9298	445717	2.3367	2418361	3.4911	4128253	1.7987	997275	2118949	1012026	3
35	1347667	0.8410	1196298	2.3832	242330	1.2704	1438628	2.0768	2786295	1.2140	745164	1359816	681311	4
36	1035744	0.6464	944751	1.8821	201315	1.0554	1146066	1.6544	2181810	0.9506	584995	1143659	453153	3
37	1414353	0.8826	2261685	4.5056	455677	2.3889	2717362	3.9227	4131715	1.8002	942089	2260273	929350	3
38	1160918	0.7245	1423261	2.8353	228104	1.1958	1651365	2.3839	2812283	1.2253	740290	1257068	714500	3
39	973719	0.6077	1215880	2.4222	189758	0.9948	1405638	2.0292	2379357	1.0367	751614	1126317	501423	3

Table 30: GCC and Espresso w/ Operating System, GCC Data

Table 31: GCC and Espresso w/ Operating System, Espresso Data

Reference Statistics:														
Total Instruction References		224015827												
Data Reads		51131704												
Data writes		12097918												
Total Data References		63229622												
Total References		287245449												
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	2709193	1.2094	3879065	7.5864	624033	5.1582	4503098	7.1218	7212291	2.5108	1226359	2782103	2334883	0
1	1813316	0.8095	2778587	5.4342	479139	3.9605	3257726	5.1522	5071042	1.7654	1020129	2561443	1489470	0
2	1004869	0.4486	1643684	3.2146	370371	3.0614	2014055	3.1853	3018924	1.0510	727587	1462206	829131	0
3	423790	0.1892	831036	1.6253	138992	1.1489	970028	1.5341	1393818	0.4852	367508	648089	378221	0
4	4735427	2.1139	6457486	12.6291	958537	7.9232	7416023	11.7287	12151450	4.2303	4391713	5365049	6291654	0
5	3774316	1.6848	4984390	9.7481	854425	7.0626	5838815	9.2343	9613131	3.3467	4430451	4710541	3142740	0
6	3300908	1.4735	4522650	8.8451	805050	6.6545	5327700	8.4260	8628608	3.0039	2098002	4737219	2145830	0
7	3372989	1.5057	6259810	12.2425	781126	6.4567	7040936	11.1355	10413925	3.6254	1040400	6025826	6257434	0
8	2573779	1.1489	4327117	8.4627	637200	5.2670	4964317	7.8513	7538096	2.6243	1245399	3000427	2746106	0
9	2270950	1.0137	3683056	7.2031	587115	4.8530	4270171	6.7534	6541121	2.2772	1307960	3850603	1374116	0
10	2607631	1.1640	7159570	14.0022	726129	6.0021	7885699	12.4715	10493330	3.6531	1036871	5943269	7554656	0
11	1857635	0.8292	4412394	8.6295	540775	4.4700	4953169	7.8336	6810804	2.3711	3053778	2598170	3331386	0
12	1698804	0.7583	3452622	6.7524	475182	3.9278	3927804	6.2120	5626608	1.9588	1022020	2919040	1685548	0
13	3584627	1.6002	4755478	9.3004	779813	6.4458	5535291	8.7543	9119918	3.1750	1165618	3456323	4497977	0
14	2450451	1.0939	3440878	6.7294	660503	5.4596	4101381	6.4865	6551832	2.2809	1319503	3124836	2107493	0
15	3055116	1.3638	3147040	6.1548	615438	5.0871	3762478	5.9505	6817594	2.3734	1308227	2309108	1847149	0
16	2633599	1.1756	4353363	8.5140	602543	4.9806	4955906	7.8379	7589505	2.6422	903109	2551165	4135231	0
17	1719156	0.7674	2824684	5.5243	484103	4.0015	3308787	5.2330	5027943	1.7504	1092781	2519996	1415166	0
18	1534490	0.6850	2591686	5.0686	447510	3.6991	3039196	4.8066	4573686	1.5923	1119073	2304776	1149837	0
19	2134566	0.9529	4629463	9.0540	525893	4.3470	5155356	8.1534	7289922	2.5379	3200611	3656088	4643782	0
20	1381642	0.6168	2644234	5.1714	393740	3.2546	3037974	4.8047	4419616	1.5386	875563	2124101	1419952	0
21	1213327	0.5416	2356703	4.6091	352674	2.9152	2709377	4.2850	3922704	1.3656	971778	2132307	818619	0
22	1700841	0.7593	3189389	6.2376	656577	5.4272	3845966	6.0825	5546807	1.9310	981054	2549900	2077891	0
23	1226111	0.5473	2192179	4.2873	512320	4.2348	2704499	4.2773	3930610	1.3684	915138	1842378	1173094	0
24	967215	0.4318	1908512	3.7325	442834	3.6604	2351346	3.7187	3318561	1.1553	757512	1388299	1172750	0
25	1209761	0.5400	2925510	5.7215	499946	4.1325	3425456	5.4175	4635217	1.6137	776435	1910807	1947975	0
26	907649	0.4052	1793422	3.5075	358141	2.9604	2151563	3.4028	3059212	1.0650	781982	1463542	813688	0
27	748531	0.3341	1469648	2.8742	290396	2.4004	1760044	2.7836	2508575	0.8733	677964	1107713	722898	0
28	924739	0.4128	3054758	5.9743	423464	3.5003	3478222	5.5009	4402961	1.5328	636823	1515429	2250709	0
29	738649	0.3297	1673561	3.2730	275992	2.2813	1949553	3.0833	2688202	0.9359	672199	1300558	715445	0
30	652803	0.2914	1321124	2.5838	221547	1.8313	1542671	2.4398	2195474	0.7643	674120	993262	488640	0
31	635991	0.2839	1791036	3.5028	309144	2.5553	2100180	3.3215	2736171	0.9526	516578	1191734	1027859	0
32	430855	0.1923	1018370	1.9917	268467	2.2191	1286837	2.0352	1717692	0.5980	472044	758295	487353	0
33	279258	0.1247	676764	1.3236	197510	1.6326	874274	1.3827	1153532	0.4016	330259	510966	312307	0
34	512841	0.2289	1845139	3.6086	253346	2.0941	2098485	3.3188	2611326	0.9091	431732	1007362	1172232	0
35	354767	0.1584	1021966	1.9987	203020	1.6781	1224986	1.9374	1579753	0.5500	443086	681871	454796	0
36	251429	0.1122	632883	1.2378	138431	1.1443	771314	1.2199	1022743	0.3561	321370	456978	244395	0
37	435774	0.1945	2205351	4.3131	254476	2.1035	2459827	3.8903	2895601	1.0081	422012	924590	1548999	0
38	329531	0.1471	1189631	2.3266	185348	1.5321	1374979	2.1746	1704510	0.5934	451474	712621	540415	0
39	252814	0.1129	736403	1.4402	121971	1.0082	858374	1.3576	1111188	0.3868	370143	503201	237844	0

Table 32: GCC and Espresso w/ Operating System, Operating System Data

Reference Statistics:														
Total Instruction References		39004710												
Data Reads		10758087												
Data writes		5592574												
Total Data References		16350661												
Total References		55355371												
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	3509603	8.9979	3187673	29.6305	846382	15.1340	4034055	24.6721	7543658	13.6277	3805303	2547478	1190752	125
1	2739339	7.0231	2366717	21.9994	704926	12.6047	3071643	18.7860	5810982	10.4976	2614797	2207422	988510	253
2	1840409	4.7184	1524595	14.1716	531326	9.5006	2055921	12.5739	3896330	7.0388	1761377	1409625	724819	509
3	895415	2.2957	996032	9.2584	300292	5.3695	1296324	7.9283	2191739	3.9594	998424	820571	372235	509
4	4695143	12.0374	3401518	31.6182	1183144	21.1556	4584662	28.0396	9279805	16.7641	4996588	2913321	1369644	252
5	4717569	12.0949	3158305	29.3575	1146195	20.4949	4304500	26.3262	9022069	16.2985	4248544	3147977	1625296	252
6	4782410	12.2611	3144080	29.2253	1099195	19.6545	4243275	25.9517	9025685	16.3050	4124353	3222608	1678472	252
7	3365821	8.6293	3634518	33.7841	895716	16.0162	4530234	27.7067	7896055	14.2643	4590232	2289144	1016554	125
8	3401375	8.7204	3207439	29.8142	839955	15.0191	4047394	24.7537	7448769	13.4563	3720265	2537925	1190454	125
9	3433106	8.8018	3158451	29.3589	812531	14.5287	3970982	24.2864	7404088	13.3756	3533983	2627527	1242453	125
10	2504572	6.4212	3813359	35.4464	811076	14.5027	4624435	28.2829	7129007	12.8786	4665061	1764009	699876	61
11	2528832	6.4834	3292318	30.6032	692307	12.3790	3984625	24.3698	6513457	11.7666	3730742	1927054	855600	61
12	2543206	6.5203	3143645	29.2212	647266	11.5737	3790911	23.1851	6334117	11.4426	3350592	1991238	992226	61
13	3760688	9.6416	2627227	24.4209	1012892	18.1114	3640119	22.2628	7400807	13.3696	3800953	2449208	1150138	508
14	3449670	8.8442	2300208	21.3812	882701	15.7834	3182909	19.4665	6632579	11.9818	2997192	2336589	1296290	508
15	3307530	8.4798	2109710	19.6105	852284	15.2396	2961994	18.1154	6269524	11.3260	2676593	2280532	1311891	508
16	2733083	7.0071	2863824	26.6202	762393	13.6322	3626217	22.1778	6359300	11.4881	3389583	2077251	892213	253
17	2577810	6.6090	2392128	22.2356	668887	11.9603	3061015	18.7210	5638825	10.1866	2454476	2116182	1067914	253
18	2540081	6.5122	2314421	21.5133	658843	11.7807	2973264	18.1844	5513345	9.9599	2280305	2136526	1096261	253
19	2053911	5.2658	2937485	27.3049	676443	12.0954	3613928	22.1026	5667839	10.2390	3140512	1831119	696083	125
20	2012465	5.1595	2631868	24.4641	540825	9.6704	3172693	19.4041	5185158	9.3670	2435207	1893700	856126	125
21	2001120	5.1305	2526561	23.4852	534767	9.5621	3061328	18.7230	5062448	9.1454	2101632	2029877	930814	125
22	2875000	7.3709	1927212	17.9141	842576	15.0660	2769788	16.9399	5644788	10.1974	2994485	1742349	906934	1020
23	2355213	6.0383	1422988	13.2271	678604	12.1340	2101592	12.8533	4456805	8.0513	2106507	1440980	908298	1020
24	2045244	5.2436	1166296	10.8411	540466	9.6640	1706762	10.4385	3752006	6.7780	1821578	1163737	765671	1020
25	2098237	5.3794	1969995	18.3118	632393	11.3077	2602388	15.9161	4700625	8.4917	2436130	1499772	764214	509
26	1762741	4.5193	1507326	14.0111	527617	9.4342	2034943	12.4456	3797684	6.8606	1654931	1369917	772327	509
27	1574483	4.0366	1333988	12.3999	435552	7.7880	1769540	10.8224	3344023	6.0410	1455471	1203412	684631	509
28	1605878	4.1171	2087116	19.4004	514052	9.1917	2601168	15.9086	4207046	7.6001	2168008	1413698	625087	253
29	1374381	3.5236	1785043	16.5926	428949	7.6700	2213992	13.5407	3588373	6.4824	1561047	1362509	664564	253
30	1297351	3.3261	1706268	15.8603	385446	6.8921	2091714	12.7928	3389065	6.1224	1358317	1358773	671722	253
31	1404643	3.6012	1422825	13.2256	547241	9.7851	1970066	12.0488	3374709	6.0964	1793254	1068173	512261	1021
32	1118686	2.8686	864790	8.0385	383814	6.8629	1248604	7.6364	2367490	4.2769	1163544	728180	474746	1020
33	924291	2.3697	605362	5.6270	271797	4.8600	877159	5.3647	1801450	3.2543	941923	524032	334474	1021
34	1075758	2.7580	1535990	14.2775	447833	8.0076	1983823	12.1330	3059581	5.5272	1630122	1001939	427011	509
35	878543	2.2524	979145	9.1015	327685	5.8593	1306830	7.9925	2185373	3.9479	996674	744608	443583	508
36	729664	1.8707	733830	6.8212	237469	4.2461	971299	5.9404	1700963	3.0728	794152	581174	325128	509
37	877237	2.2491	1793502	16.6712	397861	7.1141	2191363	13.4023	3068600	5.5435	1704283	946851	417213	253
38	731182	1.8746	1301099	12.0941	292038	5.2219	1593137	9.7436	2324319	4.1989	1031922	842595	449549	253
39	609859	1.5636	1118895	10.4005	225859	4.0386	1344754	8.2245	1954613	3.5310	832647	749839	371874	253

Table 33: GCC and Espresso w/ Operating System, Combined Data

Reference Statistics:														
Total Instruction References			423260712											
Data Reads			112087124											
Data writes			36765337											
Total Data References			148852461											
Total References			572113173											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	Int(1)	int(2)	int(3)
0	13241701	3.1285	12099406	10.7946	3021868	8.2193	15121274	10.1586	28362975	4.9576				
1	9719197	2.2963	8620998	7.6913	2221235	6.0417	10842233	7.2839	20561430	3.5939				
2	6035557	1.4260	5242959	4.6776	1478238	4.0207	6721197	4.5153	12756754	2.2298				
3	2752520	0.6503	2986134	2.6641	646650	1.7589	3632784	2.4405	6385304	1.1161				
4	18906201	4.4668	16332321	14.5711	4632931	12.6014	20965252	14.0846	39871453	6.9692				
5	17769786	4.1983	13371733	11.9298	4040250	10.9893	17411983	11.6975	35181769	6.1494				
6	17273837	4.0811	12510964	11.1618	3795905	10.3247	16306869	10.9551	33580706	5.8696				
7	13696847	3.2360	16761760	14.9542	3760378	10.2281	20522138	13.7869	34218985	5.9812				
8	12804948	3.0253	12736727	11.3632	3035696	8.2570	15772423	10.5960	28577371	4.9951				
9	12437438	2.9385	11603192	10.3519	2821780	7.6751	14424972	9.6908	26862410	4.6953				
10	10248514	2.4213	18697357	16.6811	3446816	9.3752	22144173	14.8766	32392687	5.6619				
11	9398427	2.2205	13231029	11.8042	2537333	6.9014	15768362	10.5933	25166789	4.3989				
12	9133853	2.1580	11361406	10.1362	2244937	6.1061	13606343	9.1408	22740196	3.9748				
13	14306051	3.3800	12043771	10.7450	3619356	9.8445	15663127	10.5226	29969178	5.2383				
14	12396471	2.9288	9199569	8.2075	2877269	7.8260	12076838	8.1133	24473309	4.2777				
15	12675097	2.9946	8297279	7.4025	2629763	7.1528	10927042	7.3409	23602139	4.1254				
16	10640997	2.5141	12076842	10.7745	2881433	7.8374	14958275	10.0491	25599272	4.4745				
17	9269104	2.1899	8726901	7.7858	2175851	5.9182	10902752	7.3245	20171856	3.5259				
18	8990555	2.1241	8063058	7.1936	2001557	5.4441	10064615	6.7615	19055170	3.3307				
19	8224602	1.9432	12993443	11.5923	2561203	6.9664	15554646	10.4497	23779248	4.1564				
20	7228772	1.7079	8927784	7.9650	1768730	4.8109	10696514	7.1860	17925286	3.1332				
21	7036521	1.6625	8224734	7.3378	1616866	4.3978	9841600	6.6116	16878121	2.9501				
22	9454783	2.2338	8096760	7.2236	2654766	7.2208	10751526	7.2229	20206309	3.5319				
23	7336279	1.7333	5725098	5.1077	2047270	5.5685	7772368	5.2215	15108647	2.6408				
24	6204137	1.4658	4796270	4.2791	1684761	4.5825	6481031	4.3540	12685168	2.2172				
25	7077984	1.6723	7965727	7.1067	2048987	5.5731	10014714	6.7279	17092698	2.9876				
26	5681650	1.3424	5433478	4.8475	1511639	4.1116	6945117	4.6658	12626767	2.2070				
27	5048108	1.1927	4561095	4.0692	1218248	3.3136	5779343	3.8826	10827451	1.8925				
28	5508078	1.3013	8505957	7.5887	1717105	4.6704	10223062	6.8679	15731140	2.7497				
29	4608681	1.0889	5797973	5.1727	1211078	3.2941	7009051	4.7087	11617732	2.0307				
30	4328990	1.0228	4992723	4.4543	1012395	2.7537	6005118	4.0343	10334108	1.8063				
31	4174019	0.9862	5055411	4.5103	1375094	3.7402	6430505	4.3201	10604524	1.8536				
32	3149044	0.7440	3005507	2.6814	962416	2.6177	3967923	2.6657	7116967	1.2440				
33	2327242	0.5498	2150399	1.9185	729543	1.9843	2879942	1.9348	5207184	0.9102				
34	3298491	0.7793	5353773	4.7764	1146896	3.1195	6500669	4.3672	9799160	1.7128				
35	2580977	0.6098	3197409	2.8526	773035	2.1026	3970444	2.6674	6551421	1.1451				
36	2016837	0.4765	2311464	2.0622	577215	1.5700	2888679	1.9406	4905516	0.8574				
37	2727364	0.6444	6260538	5.5854	1108014	3.0137	7368552	4.9502	10095916	1.7647				
38	2221631	0.5249	3913991	3.4919	705490	1.9189	4619481	3.1034	6841112	1.1958				
39	1836392	0.4339	3071178	2.7400	537588	1.4622	3608766	2.4244	5445158	0.9518				

Reference Statistics:															
Total Instruction References			87045931												
Data Reads			22412018												
Data writes			8521660												
Total Data References			30933678												
Total References			117979609												
Miss Statistics:															
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)	
0	569467	0.0065	3998210	0.1784	86946	0.0102	4085156	0.1321	4654623	0.0395	4457571	196924	128		
1	157381	0.0018	3621969	0.1616	53021	0.0062	3674990	0.1188	3832371	0.0325	3485561	346565	245		
2	84035	0.0010	3329636	0.1486	42786	0.0050	3372422	0.1090	3456457	0.0293	2903944	552169	344		
3	20263	0.0002	3038734	0.1356	21938	0.0026	3060672	0.0989	3080935	0.0261	2538280	542440	215		
4	1530237	0.0176	4457145	0.1989	207537	0.0244	4664682	0.1508	6194919	0.0525	5981740	212932	247		
5	27994	0.0003	4020161	0.1794	105987	0.0124	4126148	0.1334	4154142	0.0352	3932367	221519	256		
6	24653	0.0003	3891334	0.1736	82660	0.0097	3973994	0.1285	3998647	0.0339	3774844	223547	256		
7	1017717	0.0117	4762344	0.2125	284577	0.0334	5046921	0.1632	6064638	0.0514	5947827	116686	125		
8	16434	0.0002	4400027	0.1963	214112	0.0251	4614139	0.1492	4630573	0.0392	4510723	119722	128		
9	14464	0.0002	4054418	0.1809	91096	0.0107	4145514	0.1340	4159978	0.0353	4040020	119830	128		
10	1010974	0.0116	5637202	0.2515	436775	0.0513	6073977	0.1964	7084951	0.0601	7023635	61253	63		
11	10640	0.0001	5077472	0.2266	349714	0.0410	5427186	0.1754	5437826	0.0461	5374382	63380	64		
12	8924	0.0001	4421267	0.1973	249059	0.0292	4670326	0.1510	4679250	0.0397	4616166	63020	64		
13	20175	0.0002	3999361	0.1784	140605	0.0165	4139966	0.1338	4160141	0.0353	3820698	339043	400		
14	18056	0.0002	3705809	0.1653	81937	0.0096	3787746	0.1224	3805802	0.0323	3445856	359487	459		
15	13876	0.0002	3642737	0.1625	74791	0.0088	3717528	0.1202	3731404	0.0316	3361826	369092	486		
16	11935	0.0001	4205118	0.1876	165346	0.0194	4370464	0.1413	4382399	0.0371	4183986	198197	216		
17	10877	0.0001	3815142	0.1702	68491	0.0080	3883633	0.1255	3894510	0.0330	3689489	204779	242		
18	8347	0.0001	3748321	0.1672	53312	0.0063	3801633	0.1229	3809980	0.0323	3602857	206871	252		
19	7377	0.0001	4530197	0.2021	249636	0.0293	4779833	0.1545	4787210	0.0406	4677990	109106	114		
20	7196	0.0001	4047866	0.1806	112256	0.0132	4160122	0.1345	4167318	0.0353	4056470	110725	123		
21	5331	0.0001	3925458	0.1751	79899	0.0094	4005357	0.1295	4010688	0.0340	3899991	110569	128		
22	13373	0.0002	3662772	0.1634	102819	0.0121	3765591	0.1217	3778964	0.0320	3273369	505066	529		
23	8895	0.0001	3462931	0.1545	74732	0.0088	3537663	0.1144	3546558	0.0301	2993681	552286	591		
24	6951	0.0001	3422228	0.1527	72636	0.0085	3494864	0.1130	3501815	0.0297	2919004	582197	614		
25	7979	0.0001	3801405	0.1696	97760	0.0115	3899165	0.1260	3907144	0.0331	3580761	326077	306		
26	5301	0.0001	3552445	0.1585	47334	0.0056	3599779	0.1164	3605080	0.0306	3258230	346501	349		
27	4320	0.0000	3505378	0.1564	39829	0.0047	3545207	0.1146	3549527	0.0301	3190554	358605	368		
28	4933	0.0001	4027535	0.1797	131502	0.0154	4159037	0.1345	4163970	0.0353	3972352	191447	171		
29	3366	0.0000	3668285	0.1637	48399	0.0057	3716684	0.1202	3720050	0.0315	3521742	198110	198		
30	2858	0.0000	3604472	0.1608	30411	0.0036	3634883	0.1175	3637741	0.0308	3435851	201676	214		
31	5294	0.0001	3419788	0.1526	66388	0.0078	3486176	0.1127	3491470	0.0296	2997172	493952	346		
32	3147	0.0000	3283297	0.1465	39910	0.0047	3323207	0.1074	3326354	0.0282	2783241	542730	383		
33	2430	0.0000	3245695	0.1448	37795	0.0044	3283490	0.1061	3285920	0.0279	2711954	573582	384		
34	3260	0.0000	3537187	0.1578	75921	0.0089	3613108	0.1168	3616368	0.0307	3295848	320318	202		
35	2094	0.0000	3374350	0.1506	28559	0.0034	3402909	0.1100	3405003	0.0289	3062767	342009	227		
36	1628	0.0000	3329522	0.1486	22173	0.0026	3351695	0.1084	3353323	0.0284	2998066	355022	235		
37	2145	0.0000	3677102	0.1641	82525	0.0097	3759627	0.1215	3761772	0.0319	3573411	188236	125		
38	1297	0.0000	3464412	0.1546	31646	0.0037	3496058	0.1130	3497355	0.0296	3301958	195253	144		
39	1066	0.0000	3408615	0.1521	17548	0.0021	3426163	0.1108	3427229	0.0290	3227555	199525	149		

Table 34: Compress w/ Model, n=1

Table 35: GCC w/ Model, n=1

Reference Statistics:																	
Total Instruction References	160239804																
Data Reads	50197289																
Data writes	19074844																
Total Data References	69272133																
Total References	229511937																
Miss Statistics:																	
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)			
0	5705707	3.5607	3807505	7.5851	1073419	5.6274	4880924	7.0460	10586631	4.6127	10227815	358758	58				
1	3664684	2.2870	2332868	4.6474	668681	3.5056	3001549	4.3330	6666233	2.9045	6063821	602353	59				
2	2101221	1.3113	1387220	2.7635	374495	1.9633	1761715	2.5432	3862936	1.6831	3014928	847948	60				
3	911192	0.5686	763320	1.5206	139180	0.7297	902500	1.3028	1813692	0.7902	1140103	673546	43				
4	8032276	5.0127	4810068	9.5823	1812325	9.5011	6622393	9.5600	14654669	6.3851	14141175	513417	77				
5	7692272	4.8005	3407617	6.7884	1284806	6.7356	4692423	6.7739	12384695	5.3961	11829866	554741	88				
6	7557503	4.7164	2923067	5.8232	1131599	5.9324	4054666	5.8532	11612169	5.0595	11032173	579906	90				
7	6018628	3.7560	5382757	10.7232	1614482	8.4639	6997239	10.1011	13015867	5.6711	12740947	274868	52				
8	5863721	3.6593	3654970	7.2812	1018636	5.3402	4673606	6.7467	10537327	4.5912	10244733	292537	57				
9	5744545	3.5850	3112717	6.2010	882112	4.6245	3994829	5.7669	9739374	4.2435	9436741	302573	60				
10	4505734	2.8119	6352631	12.6553	1613095	8.4567	7965726	11.4992	12471460	5.4339	12328296	143129	35				
11	4403501	2.7481	4221318	8.4095	946078	4.9598	5167396	7.4596	9570897	4.1701	9420430	150428	39				
12	4293296	2.6793	3451064	6.8750	763894	4.0047	4214958	6.0846	8508254	3.7071	8353539	154673	42				
13	5384043	3.3600	3222697	6.4201	1228250	6.4391	4450947	6.4253	9834990	4.2852	9102102	732799	89				
14	4807419	3.0001	2064370	4.1125	811684	4.2553	2876054	4.1518	7683473	3.3477	6863296	820082	95				
15	4520533	2.8211	1728499	3.4434	703005	3.6855	2431504	3.5101	6952037	3.0291	6081088	870857	92				
16	4173039	2.6042	3440024	6.8530	1039724	5.4508	4479748	6.4669	8652787	3.7701	8233857	418871	59				
17	3833065	2.3921	2136034	4.2553	585332	3.0686	2721366	3.9285	6554431	2.8558	6092548	461832	60				
18	3673437	2.2925	1731504	3.4494	475062	2.4905	2206566	3.1854	5880003	2.5620	5394235	485708	60				
19	3254077	2.0308	3995681	7.9600	995051	5.2166	4990732	7.2045	8244809	3.5923	8014638	230132	39				
20	3052717	1.9051	2307772	4.5974	493996	2.5898	2801768	4.0446	5854485	2.5508	5604943	249500	42				
21	2987903	1.8646	1941179	3.8671	387231	2.0301	2328410	3.3613	5316313	2.3164	5056885	259384	44				
22	3529819	2.2028	2161430	4.3059	761631	3.9929	2923061	4.2197	6452880	2.8116	5552305	900408	167				
23	2480047	1.5477	1317053	2.6238	522367	2.7385	1839420	2.6554	4319467	1.8820	33007733	1011609	125				
24	1987837	1.2405	1155553	2.3020	497457	2.6079	1653010	2.3863	3640847	1.5863	2567827	1072917	103				
25	2774477	1.7315	2184018	4.3509	610182	3.1989	2794200	4.0337	5568677	2.4263	5009262	559320	95				
26	2016113	1.2582	1203406	2.3974	339410	1.7794	1542816	2.2272	3558929	1.5507	2932436	626410	83				
27	1695325	1.0580	1012151	2.0163	299875	1.5721	1312026	1.8940	3007351	1.3103	2341609	665675	67				
28	2198484	1.3720	2363610	4.7086	528968	2.7731	2892578	4.1757	5091062	2.2182	4757721	333277	64				
29	1711295	1.0680	1228723	2.4478	250505	1.3133	1479228	2.1354	3190523	1.3901	2822052	368415	56				
30	1541822	0.9622	955764	1.9040	197414	1.0349	1153178	1.6647	2695000	1.1742	2304260	390695	45				
31	1395316	0.8708	1374273	2.7377	378564	1.9846	1752837	2.5304	3148153	1.3717	2515499	632393	261				
32	1019655	0.6363	782571	1.5590	218108	1.1434	1000679	1.4446	2020334	0.8803	1320029	700108	197				
33	706148	0.4407	677892	1.3505	190495	0.9987	868387	1.2536	1574535	0.6860	835998	738376	161				
34	1099448	0.6861	1442121	2.8729	309928	1.6248	1752049	2.5292	2851497	1.2424	2440408	410958	131				
35	826481	0.5158	714190	1.4228	150081	0.7868	864271	1.2476	1690752	0.7367	1232145	458505	102				
36	609687	0.3805	565058	1.1257	118669	0.6221	683727	0.9870	1293414	0.5635	805730	487589	95				
37	911299	0.5687	1646866	3.2808	322521	1.6908	1969387	2.8430	2880686	1.2551	2623500	257110	76				
38	684771	0.4273	715658	1.4257	121151	0.6351	836809	1.2080	1521580	0.6630	1235086	286526	68				
39	546173	0.3408	510729	1.0174	80979	0.4245	591708	0.8542	1137881	0.4958	832022	305808	51				

Reference Statistics:														
Total Instruction References		977787939												
Data Reads		225779348												
Data writes		59867420												
Total Data References		285646768												
Total References		1263434707												
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	8834314	0.9035	12183514	5.3962	2237941	3.7382	14421455	5.0487	23255769	1.8407	21198414	2057227	128	
1	4976438	0.5089	7129086	3.1575	1441650	2.4081	8570736	3.0005	13547174	1.0722	10396821	3150124	229	
2	2601044	0.2660	4152040	1.8390	1071504	1.7898	5223544	1.8287	7824588	0.6193	4235733	3588529	326	
3	1103684	0.1129	1932512	0.8559	434592	0.7259	2367104	0.8287	3470788	0.2747	1081386	2389154	248	
4	15254790	1.5601	24436602	10.8232	4086649	6.8262	28523251	9.9855	43778041	3.4650	40924397	2853401	243	
5	9911266	1.0136	17133444	7.5886	3331877	5.5654	20465321	7.1646	30376587	2.4043	27242980	3133356	251	
6	8728055	0.8926	14070029	6.2318	2883891	4.8171	16953920	5.9353	25661975	2.0327	22395784	3285936	255	
7	10737478	1.0981	25232114	11.1756	3440440	5.7468	28672554	10.0378	39410032	3.1193	37868027	1541881	124	
8	6844904	0.7000	14971448	6.6310	2545640	4.2521	17517088	6.1324	24361992	1.9282	22693319	1668547	126	
9	5966252	0.6102	11703400	5.1836	2187945	3.6547	13891345	4.8631	19857597	1.5717	18122560	1734909	128	
10	8633747	0.8830	28947942	12.8213	3344187	5.5860	32292129	11.3049	40925876	3.2393	40103419	822393	64	
11	5252024	0.5371	16375637	7.2529	2176648	3.6358	18552285	6.4948	23804309	1.8841	22927161	877084	64	
12	4783084	0.4892	12903669	5.7152	1907117	3.1856	14810786	5.1850	19593870	1.5508	18688385	905421	64	
13	10389626	1.0626	16501205	7.3086	2856513	4.7714	19357718	6.7768	29747344	2.3545	25844954	3902015	375	
14	5063728	0.5179	10493998	4.6479	2216826	3.7029	12710824	4.4498	17774552	1.4068	13404148	4369972	432	
15	2947565	0.3015	9778940	4.3312	2025194	3.3828	11804134	4.1324	14751699	1.1676	10136118	4615125	456	
16	7539990	0.7711	15346688	6.7972	2172223	3.6284	17518911	6.1331	25058901	1.9834	22829745	2228944	212	
17	3477980	0.3557	7969160	3.5296	1535169	2.5643	9504329	3.3273	12982309	1.0275	10514555	2467514	240	
18	1921913	0.1966	6812966	3.0175	1360750	2.2729	8173716	2.8615	10095629	0.7991	7497404	2597977	248	
19	6132676	0.6272	16512600	7.3136	1962082	3.2774	18474682	6.4677	24607358	1.9477	23359046	1248194	118	
20	2775044	0.2838	7424198	3.2883	1230087	2.0547	8654285	3.0297	11429329	0.9046	10056052	1373151	126	
21	1580907	0.1617	5780615	2.5603	1028316	1.7177	6808931	2.3837	8389838	0.6640	6957297	1432413	128	
22	3915201	0.4004	10456233	4.6312	2335829	3.9017	12792062	4.4783	16707263	1.3224	12448998	4257745	520	
23	1650774	0.1688	6105799	2.7043	1644481	2.7469	7750280	2.7132	9401054	0.7441	4431695	4968775	584	
24	1050456	0.1074	5255554	2.3277	1439203	2.4040	6694757	2.3437	7745213	0.6130	2513857	5230733	623	
25	2467169	0.2523	9237184	4.0912	1700732	2.8408	10937916	3.8292	13405085	1.0610	10799613	2605164	308	
26	1009354	0.1032	4515069	1.9998	1059097	1.7691	5574166	1.9514	6583520	0.5211	3547073	3036100	347	
27	645984	0.0661	3477377	1.5402	858201	1.4335	4335578	1.5178	4981562	0.3943	1765093	3216088	381	
28	1672336	0.1710	9572162	4.2396	1463303	2.4442	11035465	3.8633	12707801	1.0058	11149394	1558218	189	
29	651389	0.0666	3775532	1.6722	756328	1.2633	4531860	1.5865	5183249	0.4103	3366527	1816515	207	
30	443912	0.0454	2479532	1.0982	543072	0.9071	3022604	1.0582	3466516	0.2744	1537449	1928837	230	
31	468053	0.0479	5918855	2.6215	1217044	2.0329	7135899	2.4982	7603952	0.6018	4938441	2665122	389	
32	322017	0.0329	3010209	1.3333	784359	1.3102	3794568	1.3284	4116585	0.3258	1223084	2893061	440	
33	208173	0.0213	2445830	1.0833	670585	1.1201	3116415	1.0910	3324588	0.2631	329944	2994176	468	
34	306048	0.0313	5767539	2.5545	946635	1.5812	6714174	2.3505	7020222	0.5556	5298258	1721717	247	
35	209095	0.0214	2358040	1.0444	531752	0.8882	2889792	1.0117	3098887	0.2453	1216734	1881874	279	
36	134902	0.0138	1695058	0.7508	406161	0.6784	2101219	0.7356	2236121	0.1770	250704	1985115	302	
37	225599	0.0231	6691741	2.9638	919504	1.5359	7611245	2.6646	7836844	0.6203	6751450	1085236	158	
38	146256	0.0150	2536024	1.1232	416869	0.6963	2952893	1.0338	3099149	0.2453	1912655	1186321	173	
39	94772	0.0097	1212266	0.5369	248621	0.4153	1460887	0.5114	1555659	0.1231	283975	1271498	186	

Table 36: Espresso w/ Model, n=1

Reference Statistics:														
Total Instruction References			5233222102											
Data Reads			1415013649											
Data writes			487428474											
Total Data References			1902442123											
Total References			7135664225											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	Int(0)	Int(1)	Int(2)	Int(3)
0	11237903	0.2147	58421027	4.1287	1585800	0.3253	60006827	3.1542	71244730	0.9984	60126100	11118508	122	
1	6312511	0.1206	40744792	2.8795	896409	0.1839	41641201	2.1888	47953712	0.6720	29872443	18081051	218	
2	2367214	0.0452	36294169	2.5649	315136	0.0647	36609305	1.9243	38976519	0.5462	14633971	24342230	318	
3	1197557	0.0229	18002413	1.2722	169793	0.0348	18172206	0.9552	19369763	0.2715	3434711	15934820	232	
4	13944663	0.2665	145937535	10.3135	1979115	0.4060	147916650	7.7751	161861313	2.2683	149865865	11995234	214	
5	13818545	0.2641	124064633	8.7677	1227917	0.2519	125292550	6.5859	139111095	1.9495	126625540	12485317	238	
6	14951105	0.2857	117123920	8.2772	1137898	0.2334	118261818	6.2163	133212923	1.8669	120557793	12654887	243	
7	10291152	0.1967	117155872	8.2795	2092895	0.4294	119248767	6.2682	129539919	1.8154	123122982	6416823	114	
8	10454930	0.1998	73426458	5.1891	1001613	0.2055	74428071	3.9122	84883001	1.1896	78268579	6614296	126	
9	10745337	0.2053	67182791	4.7479	894925	0.1836	68077716	3.5784	78823053	1.1046	72164728	6658198	127	
10	7333188	0.1401	131539713	9.2960	2792174	0.5728	134331887	7.0610	141665075	1.9853	138258290	3406723	62	
11	7142304	0.1365	52625006	3.7190	797224	0.1636	53422230	2.8081	60564534	0.8488	57081656	3482814	64	
12	7200712	0.1376	48735719	3.4442	930341	0.1909	49666060	2.6106	56866772	0.7969	53380223	3486485	64	
13	8847677	0.1691	110654476	7.8200	1135875	0.2330	111790351	5.8761	120638028	1.6906	101071917	19565769	342	
14	7835423	0.1497	95142808	6.7238	976875	0.2004	96119683	5.0524	103955106	1.4568	83117198	20837538	370	
15	4060875	0.0776	104751442	7.4029	984567	0.2020	105736009	5.5579	109796884	1.5387	88479309	21317166	409	
16	6632758	0.1267	79984777	5.6526	1199122	0.2460	81183899	4.2674	87816657	1.2307	76981951	10834512	194	
17	6606221	0.1262	52627881	3.7192	796112	0.1633	53423993	2.8082	60030214	0.8413	48655165	11374841	208	
18	5508427	0.1053	57356658	4.0534	690106	0.1416	58046764	3.0512	63555191	0.8907	52008234	11546728	229	
19	4910290	0.0938	80184487	5.6667	1746703	0.3584	81931190	4.3066	86841480	1.2170	81025985	5815387	108	
20	5042229	0.0964	32861270	2.3223	671245	0.1377	33532515	1.7626	38574744	0.5406	32536695	6037934	115	
21	4797972	0.0917	34937767	2.4691	516542	0.1060	35454309	1.8636	40252281	0.5641	34147897	6104259	125	
22	5796141	0.1108	88191763	6.2326	810913	0.1664	89002676	4.6783	94798817	1.3285	65992898	28805474	445	
23	3432362	0.0656	71980579	5.0869	582386	0.1195	72562965	3.8142	75995327	1.0650	44732561	31262267	499	
24	1374033	0.0263	71363806	5.0433	394518	0.0809	71758324	3.7719	73132357	1.0249	40715479	32416358	520	
25	4576802	0.0875	54562939	3.8560	599491	0.1230	55162430	2.8996	59739232	0.8372	42992352	16746609	271	
26	3199050	0.0611	37843460	2.6744	425751	0.0873	38269211	2.0116	41468261	0.5811	23490626	17977332	303	
27	862674	0.0165	36884932	2.6067	283468	0.0582	37168400	1.9537	38031074	0.5330	19524750	18506006	318	
28	3325275	0.0635	44512996	3.1458	969053	0.1988	45482049	2.3907	48807324	0.6840	39584436	9222725	163	
29	2469942	0.0472	20614975	1.4569	385420	0.0791	21000395	1.1039	23470337	0.3289	13649773	9820384	180	
30	1599704	0.0306	19255433	1.3608	205603	0.0422	19461036	1.0230	21060740	0.2951	11008776	10051776	188	
31	1752307	0.0335	42638700	3.0133	447914	0.0919	43086614	2.2648	44838921	0.6284	22258474	22580109	338	
32	474244	0.0091	35616617	2.5171	270932	0.0556	35887549	1.8864	36361793	0.5096	12498889	23862549	355	
33	216043	0.0041	35271621	2.4927	232540	0.0477	35504161	1.8662	35720204	0.5006	11300206	24419631	367	
34	1340261	0.0256	29395707	2.0774	577715	0.1185	29973422	1.5755	31313683	0.4388	18548940	12764531	212	
35	709074	0.0135	18726638	1.3234	205511	0.0422	18932149	0.9951	19641223	0.2753	6108357	13532639	227	
36	126862	0.0024	18441867	1.3033	173413	0.0356	18615280	0.9785	18742142	0.2627	4889262	13852639	241	
37	998025	0.0191	35824706	2.5318	750628	0.1540	36575334	1.9225	37573359	0.5266	30612516	6960711	132	
38	781205	0.0149	10394425	0.7346	154789	0.0318	10549214	0.5545	11330419	0.1588	3928174	7402100	145	
39	74918	0.0014	9802574	0.6928	127130	0.0261	9929704	0.5219	10004622	0.1402	2422673	7581796	153	

Table 37: Alvinn w/ Model, n=1

Reference Statistics:														
Total Instruction References			87045931											
Data Reads			22412018											
Data writes			8521660											
Total Data References			30933678											
Total References			117979609											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	int(0)	int(1)	int(2)	int(3)
0	570450	0.6553	3999671	17.8461	87138	1.0225	4086809	13.2115	4657259	3.9475	4451154		205977	128
1	159654	0.1834	3626167	16.1796	53406	0.6267	3679573	11.8950	3839227	3.2541	3462532		376450	245
2	87671	0.1007	3341914	14.9113	43444	0.5098	3385358	10.9439	3473029	2.9438	2838971		633714	344
3	22728	0.0261	3054661	13.6296	22571	0.2649	3077232	9.9478	3099960	2.6275	2475426		624319	215
4	1537370	1.7662	4459201	19.8965	207745	2.4378	4666946	15.0869	6204316	5.2588	5972103		231966	247
5	37711	0.0433	4022065	17.9460	106199	1.2462	4128264	13.3455	4165975	3.5311	3924907		240812	256
6	35799	0.0411	3893178	17.3709	82893	0.9727	3976071	12.8535	4011870	3.4005	3768472		243142	256
7	1021658	1.1737	4763118	21.2525	284662	3.3405	5047780	16.3181	6069438	5.1445	5945117		124196	125
8	21917	0.0252	4400605	19.6350	214197	2.5136	4614802	14.9184	4636719	3.9301	4508742		127849	128
9	20801	0.0239	4055022	18.0931	91182	1.0700	4146204	13.4035	4167005	3.5320	4038549		128328	128
10	1013030	1.1638	5637439	25.1536	436817	5.1260	6074256	19.6364	7087286	6.0072	7022694		64529	63
11	13731	0.0158	5077672	22.6560	349745	4.1042	5427417	17.5453	5441148	4.6119	5373867		67217	64
12	12571	0.0144	4421441	19.7280	249082	2.9229	4670523	15.0985	4683094	3.9694	4615783		67247	64
13	28728	0.0330	4004567	17.8679	140960	1.6541	4145527	13.4013	4174255	3.5381	3791856		381999	400
14	27785	0.0319	3710631	16.5564	82349	0.9663	3792980	12.2617	3820765	3.2385	3420120		400186	459
15	24843	0.0285	3647197	16.2734	75239	0.8829	3722436	12.0336	3747279	3.1762	3338327		408466	486
16	16837	0.0193	4207062	18.7715	165552	1.9427	4372614	14.1354	4389451	3.7205	4174716		214519	216
17	16475	0.0189	3816840	17.0303	68715	0.8064	3885555	12.5609	3902030	3.3074	3681971		219817	242
18	14469	0.0166	3749921	16.7317	53524	0.6281	3803445	12.2955	3817914	3.2361	3596305		221357	252
19	10269	0.0118	4530838	20.2161	249725	2.9305	4780563	15.4542	4790832	4.0607	4675559		115159	114
20	10547	0.0121	4048472	18.0638	112347	1.3184	4160819	13.4508	4171366	3.5357	4054605		116638	123
21	8993	0.0103	3925993	17.5174	79973	0.9385	4005966	12.9502	4014959	3.4031	3898495		116336	128
22	20219	0.0232	3675934	16.4016	103394	1.2133	3779328	12.2175	3799547	3.2205	3201153		597870	524
23	15904	0.0183	3474790	15.5041	75436	0.8852	3550226	11.4769	3566130	3.0227	2921959		643581	590
24	13939	0.0160	3433447	15.3197	73333	0.8605	3506780	11.3364	3520719	2.9842	2848085		672020	614
25	11889	0.0137	3806709	16.9851	98101	1.1512	3904810	12.6232	3916699	3.3198	3552561		363834	304
26	9263	0.0106	3557135	15.8716	47755	0.5604	3604890	11.6536	3614153	3.0634	3232812		380993	348
27	8361	0.0096	3509658	15.6597	40256	0.4724	3549914	11.4759	3558275	3.0160	3167166		390741	368
28	7188	0.0083	4029414	17.9788	131678	1.5452	4161092	13.4517	4168280	3.5331	3963111		204998	171
29	5764	0.0066	3669822	16.3743	48620	0.5705	3718442	12.0207	3724206	3.1567	3514256		209752	198
30	5361	0.0062	3605889	16.0891	30617	0.3593	3636506	11.7558	3641867	3.0869	3429492		212161	214
31	8062	0.0093	3435077	15.3269	66976	0.7860	3502053	11.3212	3510115	2.9752	2927943		581837	335
32	5327	0.0061	3296790	14.7099	40603	0.4765	3337393	10.7889	3342720	2.8333	2713818		628523	379
33	3935	0.0045	3257788	14.5359	38486	0.4516	3296274	10.6559	3300209	2.7973	2642638		657189	382
34	4880	0.0056	3543385	15.8102	76270	0.8950	3619655	11.7013	3624535	3.0722	3268293		356044	198
35	3478	0.0040	3379296	15.0781	28944	0.3397	3408240	11.0179	3411718	2.8918	3037616		373875	227
36	2583	0.0030	3333926	14.8756	22587	0.2651	3356513	10.8507	3359096	2.8472	2974677		384186	233
37	3123	0.0036	3679207	16.4162	82667	0.9701	3761874	12.1611	3764997	3.1912	3564514		200361	122
38	2111	0.0024	3466061	15.4652	31801	0.3732	3497862	11.3076	3499973	2.9666	3294350		205480	143
39	1660	0.0019	3410033	15.2152	17698	0.2077	3427731	11.0809	3429391	2.9068	3220967		208276	148

Table 38: Compress w/ Model, n=2

Reference Statistics:														
Total Instruction References			977787939											
Data Reads			225779348											
Data writes			59867420											
Total Data References			285646768											
Total References			1263434707											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	Int(0)	Int(1)	Int(2)	Int(3)
0	8847616	0.9049	12205436	5.4059	2240602	3.7426	14446038	5.0573	23293654	1.8437	21150667		2142859	128
1	5025237	0.5139	7218927	3.1973	1452993	2.4270	8671920	3.0359	13697157	1.0841	10308678		3388250	229
2	2729928	0.2792	4373868	1.9372	1102462	1.8415	5476330	1.9172	8206258	0.6495	4131300		4074632	326
3	1207257	0.1235	2102459	0.9312	460022	0.7684	2562481	0.8971	3769738	0.2984	1043624		2725866	248
4	15413408	1.5764	24472149	10.8390	4093452	6.8375	28565601	10.0003	43979009	3.4809	40760522		3218244	243
5	10074322	1.0303	17162530	7.6015	3336934	5.5739	20499464	7.1765	30573786	2.4199	27060284		3513251	251
6	8874359	0.9076	14095716	6.2431	2888223	4.8244	16983939	5.9458	25858298	2.0467	22193519		3664524	255
7	10821119	1.1067	25243653	11.1807	3442328	5.7499	28685981	10.0425	39507100	3.1270	37793297		1713679	124
8	6926166	0.7084	14980474	6.6350	2546884	4.2542	17527358	6.1360	24453524	1.9355	22605422		1847976	126
9	6039292	0.6176	11711105	5.1870	2188947	3.6563	13900052	4.8662	19939344	1.5782	18025760		1913456	128
10	8676897	0.8874	28951408	12.8229	3344592	5.5867	32296000	11.3063	40972897	3.2430	40063746		909087	64
11	5292937	0.5413	16378239	7.2541	2176894	3.6362	18555133	6.4958	23848070	1.8876	22880963		967043	64
12	4820279	0.4930	12905882	5.7161	1907368	3.1860	14813250	5.1859	19633529	1.5540	18638683		994782	64
13	10685376	1.0928	16637122	7.3688	2878670	4.8084	19515792	6.8321	30201168	2.3904	25651103		4549697	368
14	5414347	0.5537	10598703	4.6943	2235939	3.7348	12834642	4.4932	18248989	1.4444	13156604		5091955	430
15	3342512	0.3418	9869543	4.3713	2042341	3.4114	11911884	4.1701	15254396	1.2074	9869207		5384733	456
16	7701735	0.7877	15392159	6.8173	2179010	3.6397	17571169	6.1514	25272904	2.0003	22745326		2527368	210
17	3663047	0.3746	8004054	3.5451	1540633	2.5734	9544687	3.3414	13207734	1.0454	10406408		2801086	240
18	2133048	0.2182	6842999	3.0308	1365447	2.2808	8208446	2.8736	10341494	0.8185	7381267		2959979	248
19	6223307	0.6365	16526457	7.3197	1963797	3.2802	18490254	6.4731	24713561	1.9561	23316596		1396847	118
20	2877022	0.2942	7435130	3.2931	1231395	2.0569	8666525	3.0340	11543547	0.9137	10000023		1543398	126
21	1695421	0.1734	5789761	2.5643	1029388	1.7194	6819149	2.3873	8514570	0.6739	6896588		1617854	128
22	4286293	0.4384	10784401	4.7765	2388609	3.9898	13173010	4.6116	17459303	1.3819	12275038		5183771	494
23	2106687	0.2155	6462484	2.8623	1699191	2.8383	8161675	2.8573	10268362	0.8127	4238284		6029520	558
24	1578149	0.1614	5588974	2.4754	1492337	2.4927	7081311	2.4790	8659460	0.6854	2295695		6363164	601
25	2673681	0.2734	9358142	4.1448	1718547	2.8706	11076689	3.8778	13750370	1.0883	10726913		3023154	303
26	1265019	0.1294	4638482	2.0544	1076827	1.7987	5715309	2.0008	6980328	0.5525	3467498		3512496	334
27	941704	0.0963	3589791	1.5900	874830	1.4613	4464621	1.5630	5406325	0.4279	1683180		3722772	373
28	1795314	0.1836	9613292	4.2578	1468439	2.4528	11081731	3.8795	12877045	1.0192	11116485		1760373	187
29	806016	0.0824	3814893	1.6897	761190	1.2715	4576083	1.6020	5382099	0.4260	3333271		2048622	206
30	620603	0.0635	2515006	1.1139	547548	0.9146	3062554	1.0721	3683157	0.2915	1505508		2177421	228
31	663833	0.0679	6169082	2.7323	1258365	2.1019	7427447	2.6002	8091280	0.6404	4889018		3201889	373
32	529604	0.0542	3281679	1.4535	826715	1.3809	4108394	1.4383	4637998	0.3671	1171017		3466560	421
33	418346	0.0428	2736848	1.2122	715789	1.1956	3452637	1.2087	3870983	0.3064	286574		3583967	442
34	424277	0.0434	5863930	2.5972	960636	1.6046	6824566	2.3892	7248843	0.5737	5276946		1971659	238
35	333994	0.0342	2456210	1.0879	545254	0.9108	3001464	1.0508	3335458	0.2640	1196300		2138886	272
36	263202	0.0269	1797604	0.7962	420545	0.7025	2218149	0.7765	2481351	0.1964	234619		2246438	294
37	298772	0.0306	6726978	2.9794	923919	1.5433	7650897	2.6784	7949669	0.6292	6742744		1206774	151
38	224977	0.0230	2569157	1.1379	421046	0.7033	2990203	1.0468	3215180	0.2545	1905639		1309371	170
39	175120	0.0179	1245630	0.5517	252832	0.4223	1498462	0.5246	1673582	0.1325	280027		1393372	183

Table 39: GCC w/ Model, n=2

Reference Statistics:														
Total Instruction References			977787939											
Data Reads			225779348											
Data writes			59867420											
Total Data References			285646768											
Total References			1263434707											
Miss Statistics:														
Cache	Inst	%	Read	%	Write	%	Data	%	Total	%	Int(0)	Int(1)	Int(2)	Int(3)
0	8847616	0.9049	12205436	5.4059	2240602	3.7426	14446038	5.0573	23293654	1.8437	21150667		2142859	128
1	5025237	0.5139	7218927	3.1973	1452993	2.4270	8671920	3.0359	13697157	1.0841	10308678		3388250	229
2	2729928	0.2792	4373868	1.9372	1102462	1.8415	5476330	1.9172	8206258	0.6495	4131300		4074632	326
3	1207257	0.1235	2102459	0.9312	460022	0.7684	2562481	0.8971	3769738	0.2984	1043624		2725866	248
4	15413408	1.5764	24472149	10.8390	4093452	6.8375	28565601	10.0003	43979009	3.4809	40760522		3218244	243
5	10074322	1.0303	17162530	7.6015	3336934	5.5739	20499464	7.1765	30573786	2.4199	27060284		3513251	251
6	8874359	0.9076	14095716	6.2431	2888223	4.8244	16983939	5.9458	25858298	2.0467	22193519		3664524	255
7	10821119	1.1067	25243653	11.1807	3442328	5.7499	28685981	10.0425	39507100	3.1270	37793297		1713679	124
8	6926166	0.7084	14980474	6.6350	2546884	4.2542	17527358	6.1360	24453524	1.9355	22605422		1847976	126
9	6039292	0.6176	11711105	5.1870	2188947	3.6563	13900052	4.8662	19939344	1.5782	18025760		1913456	128
10	8676897	0.8874	28951408	12.8229	3344592	5.5867	32296000	11.3063	40972897	3.2430	40063746		909087	64
11	5292937	0.5413	16378239	7.2541	2176894	3.6362	18555133	6.4958	23848070	1.8876	22880963		967043	64
12	4820279	0.4930	12905882	5.7161	1907368	3.1860	14813250	5.1859	19633529	1.5540	18638683		994782	64
13	10685376	1.0928	16637122	7.3688	2878670	4.8084	19515792	6.8321	30201168	2.3904	25651103		4549697	368
14	5414347	0.5537	10598703	4.6943	2235939	3.7348	12834642	4.4932	18248989	1.4444	13156604		5091955	430
15	3342512	0.3418	9869543	4.3713	2042341	3.4114	11911884	4.1701	15254396	1.2074	9869207		5384733	456
16	7701735	0.7877	15392159	6.8173	2179010	3.6397	17571169	6.1514	25272904	2.0003	22745326		2527368	210
17	3663047	0.3746	8004054	3.5451	1540633	2.5734	9544687	3.3414	13207734	1.0454	10406408		2801086	240
18	2133048	0.2182	6842999	3.0308	1365447	2.2808	8208446	2.8736	10341494	0.8185	7381267		2959979	248
19	6223307	0.6365	16526457	7.3197	1963797	3.2802	18490254	6.4731	24713561	1.9561	23316596		1396847	118
20	2877022	0.2942	7435130	3.2931	1231395	2.0569	8666525	3.0340	11543547	0.9137	10000023		1543398	126
21	1695421	0.1734	5789761	2.5643	1029388	1.7194	6819149	2.3873	8514570	0.6739	6896588		1617854	128
22	4286293	0.4384	10784401	4.7765	2388609	3.9898	13173010	4.6116	17459303	1.3819	12275038		5183771	494
23	2106687	0.2155	6462484	2.8623	1699191	2.8383	8161675	2.8573	10268362	0.8127	4238284		6029520	558
24	1578149	0.1614	5588974	2.4754	1492337	2.4927	7081311	2.4790	8659460	0.6854	2295695		6363164	601
25	2673681	0.2734	9358142	4.1448	1718547	2.8706	11076689	3.8778	13750370	1.0883	10726913		3023154	303
26	1265019	0.1294	4638482	2.0544	1076827	1.7987	5715309	2.0008	6980328	0.5525	3467498		3512496	334
27	941704	0.0963	3589791	1.5900	874830	1.4613	4464621	1.5630	5406325	0.4279	1683180		3722772	373
28	1795314	0.1836	9613292	4.2578	1468439	2.4528	11081731	3.8795	12877045	1.0192	11116485		1760373	187
29	806016	0.0824	3814893	1.6897	761190	1.2715	4576083	1.6020	5382099	0.4260	3333271		2048622	206
30	620603	0.0635	2515006	1.1139	547548	0.9146	3062554	1.0721	3683157	0.2915	1505508		2177421	228
31	663833	0.0679	6169082	2.7323	1258365	2.1019	7427447	2.6002	8091280	0.6404	4889018		3201889	373
32	529604	0.0542	3281679	1.4535	826715	1.3809	4108394	1.4383	4637998	0.3671	1171017		3466560	421
33	418346	0.0428	2736848	1.2122	715789	1.1956	3452637	1.2087	3870983	0.3064	286574		3583967	442
34	424277	0.0434	5863930	2.5972	960636	1.6046	6824566	2.3892	7248843	0.5737	5276946		1971659	238
35	333994	0.0342	2456210	1.0879	545254	0.9108	3001464	1.0508	3335458	0.2640	1196300		2138886	272
36	263202	0.0269	1797604	0.7962	420545	0.7025	2218149	0.7765	2481351	0.1964	234619		2246438	294
37	298772	0.0306	6726978	2.9794	923919	1.5433	7650897	2.6784	7949669	0.6292	6742744		1206774	151
38	224977	0.0230	2569157	1.1379	421046	0.7033	2990203	1.0468	3215180	0.2545	1905639		1309371	170
39	175120	0.0179	1245630	0.5517	252832	0.4223	1498462	0.5246	1673582	0.1325	280027		1393372	183

Table 40: Espresso w/ Model, n=2